计算机基础教育丛书

True BASIC 程序设计

（第二版）

谭浩强　张基温　编著

清华大学出版社

JISUANJI

计算机基础教育丛书

主编 谭浩强 林定基 刘瑞挺

已出版书目

1. True BASIC 程序设计(第二版) 谭浩强 张基温
2. True BASIC 程序设计题解 谭浩强 张基温
3. FORTRAN 语言——FORTRAN77 结构化程序设计 谭浩强 田淑清
4. FORTRAN77 程序设计上机指导 朱明方等
5. FORTRAN77 结构化程序设计题解 谭浩强 主编
6. C 程序设计 谭浩强
7. C 程序设计题解与上机指导 谭浩强
8. 计算机常用算法 徐士良
9. 结构化程序设计及其在 COBOL 中的应用 谭浩强 傅金铎
10. 程序设计与开发技术 谭浩强 张基温 徐士良
11. 计算机辅助设计技术基础 孙家广
12. FoxBASE$^+$ 及其应用系统开发 史济民 主编
13. COBOL 语言 (上册)(修订版) 谭浩强

ISBN 7-302-01484-1

9 787302 014843 >

定价：15.00 元

True BASIC 程序设计

（第二版）

谭浩强　张基温　编著

清华大学出版社

(京)新登字 158 号

内 容 简 介

True BASIC 是由 BASIC 语言的两位创始人推出的新的 BASIC 版本，对 BASIC 语言作了重大的改进和发展。本书系统地介绍了 True BASIC 的结构化与模块化程序设计方法，同时通过大量例题介绍算法及如何编程，每章后均附有习题。本书采用了新的体系编写、从算法入手，使读者在学习之后能具有较强的算法设计和程序设计的能力，养成良好的程序设计风格。

在这次修订中，改正了原书中个别错处，并补充了些内容，尤其是增加了第九章文件，以适应读者需要。

本书可作为高等院校、中专和各类计算机培训班的教材，也可供自学参考。

版权所有，翻印必究。

本书封面贴有清华大学出版社激光防伪标志，无标志者不得销售。

图书在版编目(CIP)数据

True BASIC 程序设计/谭浩强，张基温编著. —北京：清华大学出版社，1994
(计算机基础教育丛书/谭浩强等主编)
ISBN 7-302-01484-1

Ⅰ. T… Ⅱ. ① 谭… ② 张… Ⅲ. BASIC 语言-程序设计 Ⅳ. TP312 BA

中国版本图书馆 CIP 数据核字(94)第 01910 号

出版者：清华大学出版社(北京清华大学校内，邮编 100084)
印刷者：清华大学印刷厂
发行者：新华书店总店北京科技发行所
开本：787×1092 1/16　印张：21.25　字数：529 千字
版次：1994 年 8 月第 2 版　1995 年 1 月第 2 次印刷
本社分类号：TP・590
印数：20001—41000
定价：15.00 元

发展計算机教育
推進四化建設

薄一波
一九九三年四月十九日

这是薄一波同志为本书作者潭浩强教授的题词，祝贺他在我国计算机教育事业中取得的成绩，勉励他作出更大的贡献。

为我国的计算机事业、赶超世界先进水平而努力！

赠谭浩强同志

张劲夫

一九九三年七月

这是张劲夫同志赠本书作者谭浩强教授的题词，祝贺他在我国计算机教育事业中取得的成绩，勉励他作出更大的贡献。

《计算机基础教育丛书》出版说明

近年来，我国的计算机应用事业迅速发展，大批科技人员、大中学生、管理人员、以及各行各业的在职人员都迫切要求学习计算机知识，他们已经认识到，计算机知识是当代知识分子的知识结构中不可缺少的重要部分。

计算机应用人才的队伍由两部分人组成：一部分是从计算机专业毕业的计算机专门人才，他们是计算机应用人才队伍中的骨干力量；另一部分是各行各业中从事计算机应用的人才。他们既熟悉本专业的业务，又掌握计算机应用的技术，人数众多，是计算机应用人才队伍的基本力量。他们掌握计算机知识的情况和应用计算机的能力在相当大程度上决定了我国计算机应用的水平。因此，在搞好计算机专业教育的同时，在广大非计算机专业中开展计算机基础教育是十分必要的。

非计算机专业中的计算机教学，无论就目的、内容、教学体系、教材、教学方法等各方面都与计算机专业有很大的不同，它以应用为目的，以应用为出发点。如果不注意这个特点，将会事倍功半。广大非计算机专业的师生、在职干部迫切希望有一套适合他们的教材，以便循序渐进地迈入计算机应用领域，并且不断地提高自己的水平。我们在前几年陆续编写了一些适合初学者使用的教材，受到广大群众的欢迎。许多读者勉励我们在此基础上进一步摸索和总结规律，为我国的广大非计算机专业人员编写一整套合适的教材。

近年来，全国许多专家、学者在这个领域作了有益的探索，写出了一批受到群众欢迎的计算机基础教育的教材。特别是全国高等学校计算机基础教育研究会作了大量的工作，在集思广益的基础上，提出了在高等学校的非计算机专业中进行计算机教育的四个层次的设想，受到广泛的注意和支持。我们认为：计算机的应用是分层次的，同样，计算机人才的培养也是分层次的；非计算机专业中各个领域的情况不同，也不能一律要求，在进行计算机教育时也应当有不同的层次。对于每一个学习计算机知识的人，还有一个由浅入深，逐步提高的过程。

我们认为，编辑出版一套全面而有层次的计算机基础教育的教材，目前不仅是十分必要的，而且是完全有条件的。在全国高等学校计算机基础教育研究会和许多同志的积极推动和清华大学出版社的大力支持下，我们决定编辑《计算机基础教育丛书》。它的对象是：高等学校非计算机专业的学生、计算机继续教育或培训班的学员、广大在职自学人员。

本丛书包括计算机科学技术的一些最基本的内容，例如计算机各种常用的

高级语言、计算机软件技术基础、计算机硬件技术基础、微型计算机的原理与应用、算法与数据结构、数据库基础、计算机辅助设计基础、微机网络与应用、系统分析与设计等，形成多层次的结构，读者可以根据需要与可能选学。

本丛书的宗旨是针对广大非计算机专业的需要和特点来组织教材，从实际出发，用读者容易理解的体系和叙述方法，深入浅出、循序渐进地帮助读者更好地掌握课程的基本内容。希望我们的丛书能在这方面具有自己的风格。在实践中接受检验。

本丛书的作者大多数是高等学校中有较丰富教学经验的教师。但是，由于计算机科学技术的飞速发展以及我们的水平有限，丛书肯定会存在许多不足，丛书的书目和内容也应当不断发展和更新。我们热情地希望得到社会各界和广大读者的批评指正。

主编　谭浩强　林定基　刘瑞挺

1988.10

前　言

近年来，计算机的应用已在我国迅速推广，计算机的知识已成为当代科技人员和管理人员的知识结构中不可缺少的一部分。为了使用计算机，人们需要学习计算机语言。BASIC 语言是一种既容易学又有广泛应用价值的计算机高级语言。前几年在我国已有近千万人学习了 BASIC 语言。不少人从学习 BASIC 语言入门，迈进了计算机应用的领域。BASIC 的突出优点是充分考虑初学者的特点，它是为初学者设计的一种计算机语言，从 1964 年 BASIC 问世以来，它以极其迅速的速度在全世界推广开来。它受到千百万初学者的热烈欢迎，在我国同样也是如此。BASIC 语言在计算机的推广普及中已经并将继续发挥巨大的作用。

在计算机科学技术迅速发展的当今，所有的语言都在不断地发展和完善，特别是要适应结构化程序设计的要求。1984 年，BASIC 语言的创始人 John G. Kemeny 和 Thomas E. Kurtz 对BASIC语言作了重大的改进和发展，提出了名为 True BASIC（意为“真正的 BASIC”）的新的 BASIC 版本，它严格遵循美国国家标准 BASIC 的规定。True BASIC 是一种比较理想的计算机语言，它不仅完全适应结构化与模块化程序设计的要求，而且保留了 BASIC 语言的优点——易学易懂，程序易编写易调试。它不仅适用于数值计算、数据处理，还有丰富的作图功能。这两位 BASIC 的创始人宣称，True BASIC 的出现将开辟 BASIC 的新纪元。

True BASIC 的出现的确给 BASIC 带来了新的生命力。学习 True BASIC 将会使读者养成编制程序的良好风格。应当大力推广 True BASIC。

为了适应发展，并且满足广大计算机初学者和爱好者的要求，我们在 1988 年编写了本书的初版，系统地介绍 True BASIC 程序设计。我们不是简单地介绍 True BASIC 语法规则，而是着眼于算法，因为算法是程序设计的灵魂，语言只是实现算法的一种工具。学习语言的目的是利用语言工具进行程序设计。本书把重点放在提高编写程序的能力上。

在写法上，本书采取科学性与通俗性相结合的方式，通过大量例题深入浅出地介绍如何使用 True BASIC 语言编写各类程序。即使从未接触过计算机的读者，也能循序渐进地学懂本书的内容。

本书初版出版后，受到广大读者的广泛欢迎，已重印 7 次。我们在这次修订中，改正了原书中一些错处，并补充了一些内容，尤其是增加了第九章文件，以适应读者需要。

马素霞同志帮助编著者进行了修订工作，对原书进行整理、修改、补充，使本书的质量得以提高。

由于我们水平有限，书中难免存在错误之处，恳请专家和读者不吝指正。

谭浩强　张基温

1993.9

目　　录

第一章　计算机算法 ……………………………………………………… (1)

1.1　算法与计算机 ……………………………………………………… (1)

1.1.1　算法的特征 ……………………………………………………… (1)

1.1.2　计算机——实现算法的有力工具 ……………………………… (6)

1.1.3　计算机科学是研究算法的科学 ………………………………… (8)

1.2　算法的表示 ……………………………………………………… (9)

1.2.1　概述 ……………………………………………………… (9)

1.2.2　流程图 ……………………………………………………… (10)

1.2.3　三种基本结构 ……………………………………………………… (11)

1.2.4　N-S 结构流程图 ……………………………………………………… (14)

1.3　用“逐步细化”方法进行算法设计 ……………………………………… (18)

1.4　程序设计语言 ……………………………………………………… (25)

1.4.1　概述 ……………………………………………………… (25)

1.4.2　程序设计语言的发展 ……………………………………………… (25)

1.4.3　程序设计语言的功能 ……………………………………………… (29)

1.4.4　程序设计语言的使用 ……………………………………………… (30)

习题 ……………………………………………………… (31)

第二章　True BASIC 程序设计初步 ……………………………………… (34)

2.1　概述 ……………………………………………………… (34)

2.1.1　True BASIC 简介 ……………………………………………… (34)

2.1.2　True BASIC 程序的组成 ………………………………………… (36)

2.1.3　True BASIC 字符集 ……………………………………………… (40)

2.1.4　使用 True BASIC ……………………………………………… (40)

2.2　数据描述 ……………………………………………………… (43)

2.2.1　数据类型 ……………………………………………………… (43)

2.2.2　常量与变量 ……………………………………………………… (44)

2.3　Ture BASIC 表达式 ……………………………………………… (46)

2.3.1　算术操作符 ……………………………………………………… (46)

2.3.2　标准函数 ……………………………………………………… (46)

2.3.3　数学表达式 ……………………………………………………… (46)

2.3.4　字符串表达式 ……………………………………………………… (49)

2.4　数据传送 ……………………………………………………… (50)

2.4.1　PRINT 语句 ……………………………………………………… (50)

2.4.2 LET 语句 …… (53)
2.4.3 INPUT 语句 …… (54)
2.4.4 DATA/READ 语句与 RESTORE 语句 …… (57)
2.4.5 数据传送语句小结 …… (61)
2.5 选取型程序结构 …… (62)
2.5.1 逻辑表达式 …… (62)
2.5.2 IF 型结构控制 …… (64)
2.5.3 CASE 型结构控制 …… (71)
2.6 循环型程序结构 …… (75)
2.6.1 DO 循环 …… (75)
2.6.2 FOR 循环 …… (84)
2.7 程序设计举例 …… (92)
2.7.1 穷举 …… (92)
2.7.2 迭代 …… (97)
2.7.3 确定性模拟 …… (107)
2.7.4 概率性模拟 …… (109)
习题 …… (119)
第三章 数组 …… (129)
3.1 用数组组织数据 …… (129)
3.1.1 几个基本概念 …… (129)
3.1.2 定义数组 …… (133)
3.1.3 数组测试函数 …… (135)
3.1.4 简单应用举例 …… (135)
3.2 数组的输入与输出 …… (147)
3.2.1 MAT PRINT 语句 …… (147)
3.2.2 MAT INPUT 语句 …… (149)
3.2.3 MAT READ 语句 …… (152)
3.3 数组赋值与运算 …… (153)
3.3.1 MAT 赋值语句 …… (153)
3.3.2 数组加 …… (155)
3.3.3 矩阵乘 …… (156)
3.3.4 数值量乘数组 …… (157)
3.3.5 内部数组常数 …… (157)
3.3.6 矩阵函数 …… (160)
3.4 排序 …… (160)
3.4.1 选择排序 …… (161)
3.4.2 插入排序 …… (163)
3.4.3 交换排序 …… (169)
习题 …… (172)

第四章　函数与子程序……………………………………………………（178）
4.1　函数 ……………………………………………………………………（178）
4.1.1　自定义函数……………………………………………………（178）
4.1.2　外部函数——使用局部变量……………………………………（186）
4.1.3　递归函数………………………………………………………（190）
4.1.4　按功能定义函数…………………………………………………（193）
4.2　子程序 …………………………………………………………………（193）
4.2.1　子程序的定义和调用……………………………………………（194）
4.2.2　内部子程序与外部子程序………………………………………（194）
4.2.3　带参子程序——虚实结合………………………………………（196）
4.2.4　递归子程序与递归程序设计……………………………………（202）
4.3　库文件 …………………………………………………………………（218）
4.3.1　库文件的概念……………………………………………………（218）
4.3.2　库文件的形成与使用……………………………………………（219）
4.4　模块化程序设计 ………………………………………………………（219）
4.4.1　设计大程序的策略——模块化…………………………………（219）
4.4.2　模块间的层次结构………………………………………………（222）
4.4.3　采用自顶向下、逐步细化的设计方法 …………………………（223）
4.5　程序的连接 ……………………………………………………………（227）
4.5.1　连接语句(Chain 语句) …………………………………………（227）
4.5.2　响应语句(Program 语句)………………………………………（228）
习题………………………………………………………………………………（228）
第五章　字符串……………………………………………………………（232）
5.1　基本概念 ………………………………………………………………（232）
5.1.1　字符串常数与字符串变量………………………………………（232）
5.1.2　字符串的比较……………………………………………………（233）
5.1.3　字符串连接与子字符串…………………………………………（233）
5.2　字符串传送 ……………………………………………………………（234）
5.2.1　用 READ/DATA 或 INPUT 语句传送字符串常数 ……………（234）
5.2.2　LINE INPUT 语句 ……………………………………………（235）
5.2.3　用 LET 语句传送字符串数据 …………………………………（236）
5.3　字符串函数 ……………………………………………………………（237）
5.3.1　测字符串长度函数………………………………………………（237）
5.3.2　字符串转换函数…………………………………………………（238）
5.3.3　字符串重复函数…………………………………………………（242）
5.3.4　子字符串查找函数………………………………………………（242）
5.3.5　删除首尾空格函数………………………………………………（243）
习题………………………………………………………………………………（244）
第六章　程序设计方法和风格……………………………………………（245）

6.1 程序质量标准 …………………………………………………………… (245)
6.1.1 关于程序质量的几个概念 ………………………………………… (245)
6.1.2 从效率第一到清晰第一 …………………………………………… (247)
6.2 结构化程序设计 …………………………………………………………… (248)
6.3 程序设计的风格 …………………………………………………………… (249)
6.3.1 基本风格:简短朴实 ……………………………………………… (250)
6.3.2 程序文档化 …………………………………………………………… (251)
6.3.3 使用过程的具体原则 ……………………………………………… (255)
6.3.4 使用控制结构的具体原则 ………………………………………… (256)
6.3.5 提高表达式的可读性 ……………………………………………… (259)
习题 ……………………………………………………………………………… (260)
第七章 数据的输入与输出 ……………………………………………… (262)
7.1 数据的输入 …………………………………………………………………… (262)
7.1.1 输入风格 ……………………………………………………………… (262)
7.1.2 单键输入(GET KEY)语句 ……………………………………… (264)
7.1.3 测试按任一键(KEY INPUT 函数) …………………………… (265)
7.2 数据输出格式的控制 ……………………………………………………… (266)
7.2.1 显示的区宽和边界 ………………………………………………… (266)
7.2.2 TAB 定位 …………………………………………………………… (268)
7.2.3 自定义输出数据项格式(PRINT USING 语句) ……………… (270)
7.2.4 MAT PRINT USING 语句 ……………………………………… (274)
7.2.5 USING $ 函数 ……………………………………………………… (275)
习题 ……………………………………………………………………………… (275)
第八章 图形 ……………………………………………………………… (278)
8.1 显示模式与图形窗口 ……………………………………………………… (278)
8.1.1 显示器的工作模式 ………………………………………………… (278)
8.1.2 图形窗口坐标 ……………………………………………………… (278)
8.2 画图 ……………………………………………………………………………… (280)
8.2.1 画点 …………………………………………………………………… (280)
8.2.2 画线 …………………………………………………………………… (281)
8.2.3 画矩形 ………………………………………………………………… (282)
8.2.4 画圆与椭圆 …………………………………………………………… (282)
8.3 着色 ……………………………………………………………………………… (283)
8.3.1 前景颜色与背景颜色 ……………………………………………… (283)
8.3.2 用 BOX AREA 语句画实体矩形 ………………………………… (285)
8.3.3 用 PLOT AREA 语句画实体图形 ……………………………… (285)
8.3.4 用 FLOOD 语句着色 ……………………………………………… (285)
8.3.5 图形中的正文设置 ………………………………………………… (285)
8.4 动画 ……………………………………………………………………………… (286)

8.5 图画 …… (286)
8.5.1 图画的定义与调用 …… (287)
8.5.2 图画的变换 …… (287)
习题 …… (288)
第九章 数据文件 …… (289)
9.1 有关概念 …… (289)
9.1.1 文件名 …… (289)
9.1.2 文件的存取方式与文件指针 …… (289)
9.1.3 文件组织 …… (290)
9.1.4 通道,文件的打开与关闭 …… (291)
9.1.5 数据文件操作的一般过程 …… (293)
9.2 正文文件操作 …… (293)
9.2.1 写正文文件 …… (293)
9.2.2 写屏幕与打印机输出 …… (295)
9.2.3 读正文文件 …… (295)
9.2.4 按数据项处理正文文件 …… (296)
9.3 记录文件操作 …… (297)
9.3.1 True BASIC 记录文件的特点及其属性设置 …… (297)
9.3.2 记录文件的存取 …… (298)
9.3.3 多数据项记录的拼装与分解 …… (301)
9.4 字节文件操作 …… (302)
9.4.1 字节文件的存取特点 …… (302)
9.4.2 字节文件应用举例 …… (304)
9.5 文件管理 …… (305)
习题 …… (306)
附录 A 编辑键一览表 …… (308)
附录 B Ture BASIC 系统命令一览表 …… (309)
B.1 文件操作 …… (309)
B.2 编辑 …… (309)
B.3 查错 …… (309)
B.4 更改行号 …… (309)
B.5 其它 …… (310)
附录 C True BASIC 语法一览表 …… (311)
C.1 简单语句 …… (312)
C.2 循环结构 …… (313)
C.3 选择结构 …… (314)
C.4 矩阵语句 …… (314)
C.5 程序单位 …… (315)
C.6 图形处理 …… (315)

C.7 文件处理 …………………………………………………………………… (317)
C.8 出错处理 …………………………………………………………………… (319)
附录 D IBM-PC 字符与 ASCII 代码对照表 ………………………………………… (321)
D.1 显示符号集 …………………………………………………………………… (321)
D.2 获得键的符号集 ……………………………………………………………… (324)
D.3 打印机符号集 ………………………………………………………………… (326)

第一章 计算机算法

电子计算机自1946年问世以来，以迅猛的速度发展，在短短的四十多年中，已经历了四个发展阶段，即所谓的“四代”——电子管计算机、晶体管计算机、集成电路计算机，大规模集成电路计算机，现在正在研制着第五代计算机。

计算机的出现是科学技术发展史上的一场伟大的革命，对人类社会产生了深远的影响。现在几乎每个领域都在大力开展计算机的应用。

计算机并不神秘，学会使用计算机也非难事。计算机是按照人们的意旨进行工作的。为了进行运算，人们必须事先准备好数据和程序。所谓程序，是指用计算机语言表示的操作步骤。例如，一个BASIC程序就是用BASIC语言写出的让计算机如何进行操作的一系列指令。

为了编好程序，就必须事先整理出解题的思路，正确地拟定出每一个操作步骤。“算法”就是研究这个问题的。本章将介绍算法、计算机和程序的有关知识。

1.1 算法与计算机

1.1.1 算法的特征

初学者往往认为计算机解题是一个不可思议的过程。其实，计算机解题的过程与操作机器、执行任务、演奏音乐、打算盘、打太极拳、炒菜、做饭等极为类似。

广义地说，做任何事情都是一个过程，这个过程可以看作为在规定条件下能够进行的基本操作所组成的序列。例如，开汽车的过程可以看作是如下一个可进行的基本操作序列：

1. 掏出钥匙；2. 打开车门；3. 坐在座位上；4. 打开电门；5. 打火；6. 加油；7. 踩离合器；……。或者说，我们在做任何事之前，都要在规定的解题环境之内，用所允许的基本操作去构造解题的步骤，只不过在做那些我们已经熟悉（习惯）的工作时，无须再有意识地考虑它罢了。例如，歌手唱歌、乐队演奏，都要为他们事先设计乐谱，乐谱就是由音符组成的序列，而音符就是音乐中所规定的基本操作的符号。同样，打太极拳的图谱、做菜的菜谱、珠算的口诀、工作计划、生产流程、治病处方、……，都代表了在不同的解题环境中，为实现某一目的所应完成的基本操作序列。只不过在不同的解题环境下，对基本操作有不同的定义。这种解题操作序列称为“算法”。这里“解题”二字是泛指解决某一问题，而不只是指“计算”。计算机科学家D. E. Knuth说：“一个算法是一个有穷规则的集合，其中的规则规定了一个解决某一特定类型问题的解答。”算法的概念源于数学，当然包括了在数学中的应用。

例1.1.1 有三个数a,b,c，找出其中最大的数。

我们先考虑处理这类问题的思路，先将a和b两数相比，将大者放在变量max中，然后再让c与max相比，如果c>max，将c的值送到max中，最后max就是三个数中最大的数。将它写成以下形式：

S1：将 a 与 b 比，如是 a>b，则使 a 的值作为 max 的值，否则，使 b 的值作为 max 的值。

可以写成：如 a>b，则 a⇒max，否则，b⇒max。

S2：将 c 与 max 比，如 c>max，则 c⇒max。

最后 max 的值就是三个数中最大的数。

上面用 S1，S2 表示步骤的次序，在写算法时常用这种形式的标记。S 是 step（步）的首字母。S1 代表"第一步"，S2 代表"第二步"，……。

例 1.1.2 有两个变量 a 和 b，要求将它们的值互换。例如 a=3，b=4，互换后，a=4，b=3。

为了进行两个变量的值互换，须引入一个临时变量。正如两个瓶子内的液体互换需要用第三个瓶子作为过渡一样，见图 1.1。其算法可表示如下：

S1：a⇒c（将 a 的值送给变量 c）；

S2：b⇒a（将 b 的值送给变量 a）；

S3：c⇒b（将 c 的值送给变量 b）。

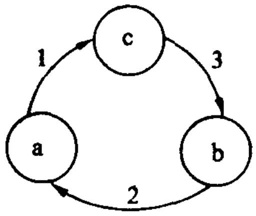


图 1.1

通过以上三个步骤实现了两个变量值的交换。这个方法在以后程序设计中常会用到。

例 1.1.3 求 1+2+3+4+5，即求 $\sum_{n=1}^{5} n$。

先用最原始的方法进行：

S1：先进行 1+2 的运算，相加的和（结果）放在变量 N 中（N 的值为 3）。

S2：将 N 再加 3，和仍放在 N 中（此时 N 的值为 6）。

S3：将 N 的值再加 4，和仍放在 N 中（此时 N 的值为 10）。

S4：将 N 的值再加 5，和仍放在 N 中（此时 N 的值为 15）。

可将算法写成如下形式：

S1：令 N=1

S2：令 I=2

S3：N 与 I 相加，结果放在 N 中

S4：使 I 的值加 1，即 I+1⇒I

S5：如果 I≤5，返回 S3，否则算法终止

最后 N 的值就是 $\sum_{n=1}^{5} n$ 的结果。

在执行这个算法时，S3 和 S4 两个步骤要重复执行多次。直到 I>5 为止（请读者考虑 S3 和 S4 重复执行多少次？）。可以看到，这个算法是采用循环的方法来处理的。请考虑：在 S5 步骤中的循环继续的条件为什么是"I≤5"，如果改写为"I<5"会出现什么情况？

显然，这个算法是比较好的。如果需要求的是 $\sum_{n=1}^{100} n$，即 1+2+…+100，上述算法基本上可不必改，只需将 S5 步骤中的"若 I≤5"改为"若 I≤100"即可。读者可自己写出此完整的算法，并分析它的执行过程。

例 1.1.4 有 10 个数，找出其中最大的数。

本题的思路：如同打擂台一样，台上先站着一个人，第二个人上台与之比较，胜者留在台上，然后第三个人上台再与台上的人（胜者）比较，胜者留在台上，比完九轮之后，在台上的人

便是最后胜者。

今先将输入的第一个数放在MAX中，将输入的第二个数与之相比，如第二个数大于MAX，则它取代MAX的原值。然后第三个数再与MAX比，大者放在MAX中……，一直到比完9次为止。可以写成以下的算法：

S1：输入一个数，放在MAX中。

S2：1⇒I(I用来累计比较次数)。

S3：输入一个数，放在X中。

S4：若X>MAX，则X⇒MAX。

S5：I+1⇒I(I的值加1)。

S6：若I≤9，返回S3继续执行。否则，停止。此时MAX的值就是10个数中最大的数。

例1.1.5 求两个正整数m和n(m>0,n>0,m>n)的最大公约数。

求两个正整数的最大公约数的算法是介绍算法时常常引用的一个典型例子。为了使读者容易理解，我们首先看看手工求M=60和n=33的最大公约数的过程：

S1：以n(33)除m(60)，得余数r(27)。

S2：判断r是否等于零：若r=0，则n为解，若r≠0，则进行S3的操作(现r=27，故进入操作S3)。

S3：以n作为新的m(33)，以r作为新的n(27)求新的m/n的余数r(6)。

S4：判断r是否为零：若r=0，则前一个n即为解；否则要继续S5的操作。

S5：以n作为新的m(即使27⇒m)，以r作为新的n(即使6⇒n)，求新的余数r(3)。

S6：判断上一个r是否等于零：若r=0，则前一个n即为解；否则执行下面的操作。

S7：再以n作为新的m(6⇒m)，r作为新的n(3⇒n)，求新的r(r=0)。

S8：判断上一个r是否等于零。这里已有r=0，所以算法结束，n=3即为60与33的最大公约数。

这种算法称为辗转相除。算法中使用的基本操作：整除求余和判断。它们都是进行算术运算时所允许的操作，只不过判断这一操作是在人脑中完成的，并不需要显式地写出来罢了。

从上面的算法中可以看出，S8,S6,S4与S2所完成的操作是相同的，S7,S5与S3所完成的操作也是相同的。或者说，S8,S6,S4是S2的重复，S7,S5是S3的重复。经过加工整理，可以得到下面的一个简单明了的算法2：

S1：置数。将两个数中的大数放到m中，小数放到n中。

S2：求余。求m/n的余数r。

S3：判断。若r=0，则n就是所求最大公约数；若r≠0，执行下一步。

S4：置换。使n值作为新的m值(n⇒m)，使r值作为新的n值(r⇒n)。

这个算法的S4，使用了循环，控制S2,S3,S4重复执行几次，使算法变得简单明了。稍作变换，我们还可以将它变换为下面的算法3：

S1：置数。将两个数中大数放到m中，小数放到n中。

S2：重复执行下面的序列，直到求得r=0为止。

S2.1：求余。求m/n的余数r。

S2.2：置换。n⇒m,r⇒n。

S3：输出 m。

上面的算法对任何 m 与 n 都是适用的。要理解这个算法或要写出这个算法需要经过一定的抽象，即从具体问题出发，经过提炼、归纳，找出解决本类问题的普遍规律。譬如在智力竞赛时，对于“1+2+…+100”这样一个题目，有人按①作 1+2=3，②作 3+3=6，③作 6+4=10，……的方法去作；而有人找到了另一种高效率的算法：100+(1+99)+(2+98)+…+(49+51)+50，共得 50 个 100，1 个 50，即和为 5050。一旦得到这样一种方法，当遇到题目为 1+2+…+1000 时，他决不会重新冥思苦想地另寻算法，而是运用前面的方法，很快算出 1000+(1+999)+(2+998)+…+(499+501)+500=500500。

例 1.1.6 求正整数 A 和 B 的积，A 和 B 为任意位数。

先看一下人工计算是怎样进行的。假如 A=3412，B=513，乘法竖式为

```
      3413——A
     ×513——B
    ———————
     10236 ┐
     3412  ├部分积
   17060   ┘
   ———————
   1750356——积
```

它们是这样进行的：以 A 为被乘数，分别以 B 的个位数、十位数、百位数作为乘数，求出它们的部分积。个位数与 A 相乘，得到第一行部分积，十位数与 A 相乘得到部分积(第二行)应左移一位，或者认为将该位数字乘以 10 再和 A 相乘；同样由乘百位数得到的第三行部分积应左移二位，或者认为将该数乘以 100 再和 A 相乘。把各部分积相加即得最后的积。

现在 A 和 B 的位数都是任意的，那么怎样用一般抽象的方法表示此算法呢？

将乘数 B 表示为 $b_nb_{n-1}\cdots b_2b_1b_0$，$b_0$ 表示个位数字，b_1 表示十位的数字，b_2 表示百位的数字，其余类推。用 C 表示某一时刻的部分之和。用 i 表示乘数的位数，i=0 表示个位(因为 $10^0=1$)，i=1 表示十位(因为 $10^1=10$)，i=2 表示百位(因为 $10^2=100$)，……。

算法可以表示如下：

S1：使 i=0，C=0(未进行乘数运算前，部分积之和为零)，并给 n 赋初值。

S2：当 i≤n 时，重复执行下面的操作：

S2.1：$C+A\times b_i\times 10^i\Rightarrow C$(乘 10^i 即左移 i 位，第一个部分积为 $0+A\times b_0\times 10^0=A\times b_0$)；

S2.2：$i+1\Rightarrow i$(i 增 1)。

S3：输出 C。

请读者按上述算法进行 853×28654 的运算。

例 1.1.7 求两个整数 X(X≥0)和 Y(Y>0)的整数商和余数(规定只能用加法和减法运算)。

除法运算实际上是用加法和减法来实现的。其办法是，从 X 中一次又一次地减去 Y，直到 X<Y 为止，所减的次数即为商。例如 5÷2 的商为 2，余数为 1。可以将 5 先减 2，余 3，再减一次 2，余 1。每减一次，商加 1。5 能减两次 2，所以商就得 2。此时余数为 1，不能再减 2，结束。用 Q 代表商，R 代表余数，算法可以表示为

S1：使 Q=0，R=X(在开始减以前，商为零，所余的数就是 X 本身)。

S2：当 R≥Y 时，重复下面的操作：

S2.1：R－Y⇒R；

S2.2：Q＋1⇒Q。

S3：输出 R。

上面我们举了几个数学算法的例子。每一个算法都由加、减、乘、除、判断、置数等基本操作，按顺序、判断、分支、重复等结构组成。一般说来，任何复杂的数学问题都可以由上述这些最基本的操作按一定的结构组成的算法来求解。研究算法就是研究怎样把各种类型的问题的求解过程分解成上面的基本操作。

实际上，算法可以看作与演绎平行的另一数学体系。

由前面的讨论可知，算法是对解题过程的抽象和精确描述。一般来说，一个算法应当具备以下几个特征：

1. 有穷性：一个算法应当包括有限个操作步骤，或者说它是由一个有穷的操作系列组成，而不应该是无限的。例如，创作一首小提琴曲谱可以称为小提琴算法。如果有人说他设计一首永远演不完的小提琴曲谱，人们将无法演奏完这个曲子，因而它不能称为算法。

不仅如此，有穷性还常常理解为实际上能够容忍的合理限度。例如，执行一个计算机算法需要让计算机运行 10000 年，便是不能容忍的。

2. 确定性。算法中的每一步的含义都应该是清楚无误的，不能模棱两可，也就是说不应该存在"歧义性"(即可作两种以上不同的解释)。请读者分析下面这句话：张三对李四说他的孩子考上了大学。这句话就是"歧义性"的，可以理解为张三的孩子考上了大学，也可理解为李四的孩子考上了大学。因此在表示算法时要使用明确的文字或数字语言。

3. 一个算法应该有零个或多个输入。如例 1.1.5(求最大公约数)中，有两个输入量 m 和 n。有的算法也可以没有输入。例如一盒音乐磁带，只要启动，就会放出音乐，而没有输入。

4. 一个算法应该有一个或多个输出。算法的目的是求解，"解"就是输出。对无解的问题也应给出"无解"的信息。没有输出的算法是没有意义的。例 1.1.5 中的输出是最大公约数。

5. 有效性。即我们在前面已经提到的一个算法必须遵循特定条件下的解题规则，组成它的每一个操作都应该是在特定的解题规则中允许使用的、可以执行的，并且最后能得出确定的结果。如在例 1.1.5 中给定的 n 为零时，执行第一步便会遇到一个在算术运算的条件下是无意义的运算(分母为 0)。只要算法中有一个操作是不可执行的，整个算法就不具有有效性。

基于算法的特征，还可以给出算法的另一种定义：算法是一个过程，这个过程由一套清楚的规则组成，这些规则指定了一个操作顺序，以便用有限的步骤提供特定类型问题的解答。这里需要指出的是，并非所有的过程都能称为"算法"。过程可以不受"有穷性"的约束，即它的操作可以是无穷尽的。例如求所有自然数之和，它要执行无穷尽的加法，那么这个工作便是一个"过程"，而不能称为算法。

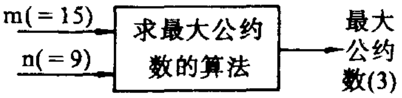


图 1.2

对使用算法的人(而不是设计算法的人)来说，可以把一个算法看作一个"黑箱"，给它输入某些确定的量，它就会输出确定的输出量。如求 m 和 n 最大公约数的算法的"外部特性"可以用图 1.2 表示。也就是说，从算法的外部(而不是研究算法的内部)看，它的作用是：给它输入 m 和 n，

它就能输出一个最大公约数。从此例也可以理解算法的输入和输出的概念。

1.1.2 计算机——实现算法的有力工具

前面我们讨论了算法的基本概念，但是算法只是指出操作的内容和步骤，或者说只给出了实现目标的过程描述，要把它付诸实践得到预期结果，还需要解决工具问题，即用什么工具去实现算法。正如用珠算算题，口诀是算法、算盘是工具一样。只有口诀没有算盘，就难以有效地算题。再如，作曲家写出一首名曲，用五线谱形式发表，这就是算法。但是五线谱不会发声，需要某种乐器，按照乐谱演奏出来，这就叫做“算法的实现”。最古老的算法工具是人工方式。用人工方式处理一些简单的算法，还能在可容忍的时间内完成，而对于复杂的算法，则往往人工难以在可容忍的时间范围内完成。例如，国外有人用手工计算 π 值，计算到小数点后的 707 位，花了十五年时间。如果要计算到小数点后的 5000 位，恐怕一辈子也完不成。

人类从远古时代便开始制造工具，以放大或延伸自身的能力。计算工具是一种智力工具。从棍石记事、屈指计算开始，人类先后制造出了算筹、算盘、计算尺、手摇计算机、电动计算机等计算工具，用以“放大”自身的智能，提高智力劳动的效率。1946 年，世界上第一台电子计算机 ENIAC 诞生了。这是当时已经相当成熟与丰富的计算理论与电子技术相结合的硕果，也是当时正在迅猛发展的科学技术的迫切需要。ENIAC 是在美国陆军部主持下，由宾夕法尼亚大学电子系工程师 J. P. Eckert 和物理学家 J. W. Mauchly 等人研制成功的。它的运算速度比当时已有的其它类型的计算机提高了 1000 倍，每秒钟可进行 5000 次加法、300 次乘法。40 年代末以来，电子计算机一直在突飞猛进地发展，已经经历了电子管、晶体管、集成电路、大规模集成电路等四代。运算速度已可达到每秒亿次以上。这样高的运算速度使其之前的任何计算工具及人的智力所望尘莫及，也使得我们能在允许的时间内完成过去无法实现的许多算法，如前述要把 π 的值计算到小数点后面 5000 位等问题，都可以在短时间内实现。

然而，电子计算机的优势还并不单单在于其运算速度高上。它区别于其前那些计算工具的另一惊人成就是它能自动地执行操作。这一成果主要归功于一位美籍匈牙利数学家冯诺曼(Von Neumann)。

1944 年夏天，正在普林斯顿工作的冯诺曼偶然地得到了正在研制 ENIAC 的消息，并引起了他的极大兴趣。他很快发现 ENIAC 的致命缺点，提出了以二进制和程序存储控制为基础的计算机体系结构思想。

二进制原理以二元逻辑为基础。它要求所有信息在机器内部都用 0,1 两个码的组合表示。表 1.1 是几个普通十进制数的 0,1 码表示。它遵循逢二进一的规律，也称二进制数。二进制原理确立了电子计算机经济而实用的物理结构。

程序存储控制理论使算法的自动执行成为可能。程序是以机器能理解的信息描述的算法。以前，程序被保存在机器外部，机器要由人按程序的规定步骤一步一步地操作，如用算盘进行计算时，每一步运算都要由人按口诀内容拨动珠子进行。这就把人与机器捆在了一起，不仅把人束缚在繁冗的运算过程之中，而且难以发挥出机器运算速度高的这一突出优势。程序存储控制原理就是要让机器“记住”程序，并按程序规定的步骤控制自己的工作。这样便可以不需要人再去干预运算过程，解放了人，同时使机器的运算速度高的优势得到充分发挥。

现代计算机基本上都是按冯诺曼原理制造的，所以也称为冯诺曼型计算机。按照程序存

表 1.1

十进制数	二进制数	十进制数	二进制数
0	0	5	101
1	1	6	110
2	10	7	111
3	11	8	1000
4	100	9	1001

储控制原理，现代计算机应由输入设备、输出设备、运算器、控制器、存储器等五大部件组成，如图 1.3 所示。当人用输入设备把程序和数据送到机器后，机器就把它们保存到存储器中。执行时，控制器从主存储器中依次取出程序指令，并逐条进行分析，根据指令的要求控制运算器对指定的数据进行相应的运算。控制器和运算器是计算机的心脏，由它进行控制与操作，它称为中央处理器(Centre Processing Unit)，简称 CPU。

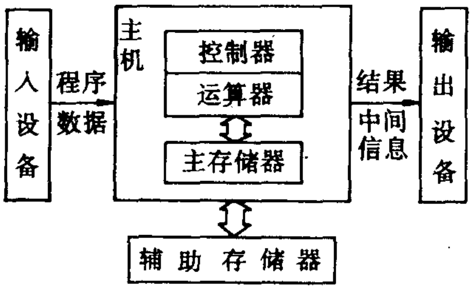


图 1.3

输出设备能将运算的中间信息(如出错、中间结果等)和最终结果输出到某个输出设备上。常用的输入设备是键盘，输出设备是显示器和打印机。

在计算机的全部设备中，存储器是一个很重要的设备。就像人没有记忆功能，就什么也做不成一样，计算机没有存储器，就无法存储程序和数据，就不能进行程序存储控制，自动工作也就无法谈起。

二进制电路是存储器的“细胞”。每个“细胞”可以存储一个二进制信息码，要么是“1”，要么是“0”，称为一“位”(b，即 bit 的缩写。一个存储器可以存储亿万个二进制信息。为了管理方便，现代计算机把几个二进制信息组织在一起，称为一个“字节”(B，即 byte 的缩写)。一般一个字节由八位组成。存储器的容量一般以字节为单位，如长城 GW0520DH 的存储器可供用户使用的容量为 640KB(1K$=2^{10}=$1024)，而长城 GW286B 为 1MB(1M$=2^{20}$，约为 1 兆)，IBM PC 为 64KB，IBM PS/2-30 为 640KB，IBM PS/2-80 为 2～16MB。计算机中的一个逻辑信息，即一条指令或一个数据，称为一个计算机字(word)。一个字所包含的二进制位的个数称为字长。通常字长是字节的整数倍。如 True BASIC 规定每个数值数据占 8 个字节。

多数计算机是以字节为单位进行信息存储的。存储一个数据的存储空间，称为一个存储单元。一个存储单元可以包括一个字节或几个字节。为了能向指定的存储单元存入或取出指令或数据，需要对每一个存储单元编号，如同旅馆中的房间号一样。存储单元的编号，称为该单元的“地址”。按地址检索信息，是当代计算机存储器的一大特征。即向存储器中存入一个信息时，应指出把该数据存入哪个单元(指出地址)，而从存储器中取一个数据时，是指取出哪个单元内的数据，而不是指取出哪个数值。如图 1.4 所示，假设在 2001 单元内已放入了数据“2”，在 2002 单元中已放入了数据“3”，如果想进行 2+3 的运算，应说“将 2001 单元中的内容与 2002 单元中的内容相加，送到 2003 单元中”，记作

$$(2001)+(2002)\Rightarrow 2003$$

(2001)表示 2001 单元中的内容，而 2003 指单元地址而不是内容。请读者不要混淆单元

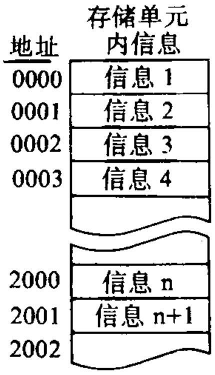


图 1.4

地址与单元内容。正如不要混淆旅馆中的某房间与该房间中的旅客一样。上式中不加括弧的数字指的是单元地址，加括弧的指的是单元内容。一个存储单元中存入一个信息后，只要不再向它里面存入别的信息，原有的信息便会一直保存，可供多次“取用”，其值不变。就像在磁带中录入一首歌曲后，只要不用它录制别的歌曲，原来的歌曲便会一直保存，可供反复欣赏一样。而一旦重新存入了别的信息，原来的信息便不复存在了。如进行这样的操作：

$$(2001)+(2002)\Rightarrow 2001$$

操作结束后，2001 单元内的数据便不再是 2，而成为 5 了。存储器的这一特点可通俗地称为“取之不尽，新来旧去”。这种存取与向一个仓库中存取货物有本质区别(仓库中每经一次存取之后，物品的数量是变化的，而从存储单元中“取数”后，单元中内容不变)，因此，把信息的存取称为“写入”或“读出”更为确切、形象。

容量、存取速度和成本是评价计算机存储器功能的三项主要指标。但这三项指标往往是互相矛盾的。例如，容量大，存取时为寻找某一单元所花费的时间就多；成本高，就难以做得容量很大。为此，现代计算机采用“分级存储”技术来解决这一矛盾。通常采用两级存储方式，即把存储器分为主存储器(即内存储器)和辅助存储器(即外存储器)两级。对内存的存取速度高，通常用以存储当前需处理的信息。对外存的存取速度低，但价格便宜，可以做得容量很大。计算机工作时，内存与外存之间可以成批交换数据。目前内存基本上采用半导体存储器，外存多用磁带、磁盘等磁表面存储器。半导体存储器是一种“易失型”存储器，断电后原来存储的信息会全部丢失。而磁介质存储器的信息则不会因断电而消失。可以长期保存信息。顺便说明一下，计算机存储器中有成千上万个存储单元，要在程序设计时给指令和数据一一分配存储单元是一件十分烦琐的工作。这项工作现在已经可以交给一种软件——操作系统去完成了。这样，程序员就可以有可能用一个名字(即变量)来代替地址编写程序了。这样 2+3 便可以写成

$2\Rightarrow A$

$3\Rightarrow B$

$A+B\Rightarrow C$

其中“⇒”代表往存储单元中送数据。

1.1.3 计算机科学是研究算法的科学

输入设备、输出设备、运算器、控制器、存储器分别承担了计算机的输入、输出、运算、自我控制和记忆等功能。然而这些物理部件(也称计算机硬件)只不过是机器的躯体。计算机的全部工作实际上是在程序的控制下进行的。或者说这些物理部件是在程序的支配下工作的。所以人们常把程序称为计算机的灵魂。然而，这个灵魂是人给的，人就是计算机的“上帝”。

计算机程序就是用计算机能够理解的信息(即计算机语言)描述的算法。为了让计算机正确、可靠而高效地完成解题任务，需要设计一个好的算法，然后根据此算法编写程序。因

此，从某种意义上说，算法的研究就成为计算机科学的核心。当然，这并不是说，只有有了计算机才有了算法科学。其实算法是一门很古老的科学。

在古代，公理系统还没有发展起来的时候，人们只会利用算术方法解题。不论什么问题都要用算术方法求解，这就迫使人们不得不去精心地构造各种算法，使得以算术为代表的算法科学曾一度得到了高度发展。但是，随着公理系统的发展，人们把注意力和兴趣渐渐地转向以公理系统为基础的、内容丰富的、结构复杂的演算，而把算法——这种在人类史上创造过灿烂的文明之光的数学思想看作是“笨”办法逐步打入冷宫。

随着自动机器，尤其是电子计算机的出现，人们又开始注意算法的研究，使算法获得了新生。

由于计算机运算的速度极高，而且从根本上来说计算机只能执行一些最简单的操作（计算机所能完成的所有复杂的运算都是由一系列简单的操作组成的），因此人们必须利用计算机高速操作这一特点去完成一系列简单而“笨”的操作。这就不得不重新认识算法在现代条件下的价值，深入地进行算法的研究。我们面临的任务：用计算机能够执行的基本操作去组成解题步骤——算法。或者说，是要把一个算法分解为计算机可以执行的基本操作。

计算机虽然是用现代科学技术武装起来的高技术产品，它的基本部件却是只能表示 0 或 1 的电子开关。计算机能执行以下的基本操作：

1. 数据传送：由输入设备把数据送到内存；由内存把数据传送到输出设备；由内存一个单元把数据传送到另外一个单元。
2. 逻辑运算：“与”、“或”、“非”。
3. 算术运算：加、减、乘、除、乘方等。
4. 比较、判断和转移。

要靠这些简单的操作的组合去解决复杂问题，迫使人们不得不花很大的力气去研究如何用计算机解题的计算机算法。

通常把计算机处理的问题分为两大类：一类是“数值运算”，如解一个联立方程；求一个函数的定积分等。另一类是：“非数值运算”，例如，有一批人名要按它们的汉语拼音字母排序；用计算机检索图书馆中的书刊等。过去有的人以为计算机的作用只是进行数值计算，这是一个误解。当今，计算机已深入到人类生产和生活的各个领域，在非数值领域的应用远远超过了数值运算领域。50 年代初期，计算机在数值计算方面的广泛应用，使得讨论计算方法一类算法的计算数学迅速发展并成熟。50 年代中后期，计算机开始广泛应用于非数值运算领域。非数值运算的特点是其处理对象大多数是非连续性的数据（例如图书馆中书目数据是不连续的），这种数据称为“离散性”数据。非数值运算领域的应用促进了以研究离散对象为目标的组合算法的研究。

1.2 算法的表示

1.2.1 概述

前面述及，算法是解题方法的精确描述。描述算法的工具对算法的质量有很大的影响。一般说来，可以使用三种类型的描述工具来表示算法。

1. 自然语言:人们日常使用的语言,如汉语、英语、日语等。上节中的例子基本上都是用汉语描述的算法。使用自然语言描述算法,人们比较习惯,容易接受,但它有以下缺点:

(1) 自然语言容易有“歧义性”,不太严格。例如前面说到的“张三对李四说他的孩子考上了大学”这句话就不严格,具有歧义性。

(2) 用自然语言描述比较冗长。例如:“置 m 的值为 n,置 n 的值为 r”,不如表示成“n⇒m,r⇒n”简洁。

(3) 自然语言的表示形式是顺序的,即一句一句地顺序地叙述下来,或者说各步骤是串行的。如果算法中有分支或转移时,用文字表示就显得不那么直观。

因此,自然语言不是一种好的算法描述工具,只有在不致引起混乱的情况下,才可以用简洁的文字来描述某一步骤的操作。

2. 专用工具:为了更好地、准确地描述算法,人们创造了许多专用的算法工具。这些算法工具有使用图形的(如流程图,结构化流程图,PAD 图,IPO 图,Warnier 图等),也有使用表格的(如判定表等),还有使用代码符号的(如伪代码)。

3. 计算机程序设计语言:计算机程序设计语言是可以让计算机接受、理解和执行的算法描述工具。任何计算机算法最终都要用程序设计语言描述出来,才能在计算机上实现。用计算机程序设计语言描述的算法的形式也是串行的。串行形式难以直观地反映算法的逻辑结构,例如当有转移、分支、循环时,直接看程序就不那么直观,所以在算法设计阶段人们多用专用描述工具。下面介绍两种常用的算法的描述工具——流程图和 N-S 结构化流程图。

1.2.2 流程图

流程图是一种算法的图形描述工具。它用一些几何图框表示各种类型的操作,在框内写上简明的文字或符号表示具体的操作,用箭头的流程表示操作的先后顺序。图 1.5 为 ANSI(美国国家标准化协会)规定的一些常用的流程图符号。

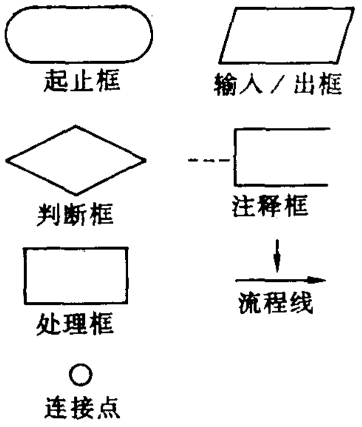


图 1.5

下面是用流程图表示算法的例子。

例 1.2.1 用流程图表示例 1.1.1 的算法(找出三个数中最大的数),见图 1.6。

与例 1.1.1 所列的算法相比,增加了“输入 a,b,c”和输出 max 值的两个框,以适应计算机算法的特点。

例 1.2.2 用流程图表示例 1.1.2 的算法(两数互换),见图 1.7

例 1.2.3 用流程图表示例 1.1.3 的算法 $\left(求\sum_{n=1}^{5}n\right)$,见图 1.8。

例 1.2.4 用流程图表示例 1.1.4 的算法(找 10 个数中的最大数),见图 1.9。

例 1.2.5 用流程图表示例 1.1.5 的算法(求最大公约数),见图 1.10。

在图 1.10 中用了一个“注释框”,用来声明“m,n 应为正整数”。

例 1.2.6 用流程图表示例 1.1.6 的算法(求 A 和 B 的乘积),见图 1.11。

输入 A,B 的值和 B 的位数,例如,B=513,则 m=3。n=m−1=2。B 可以表示为

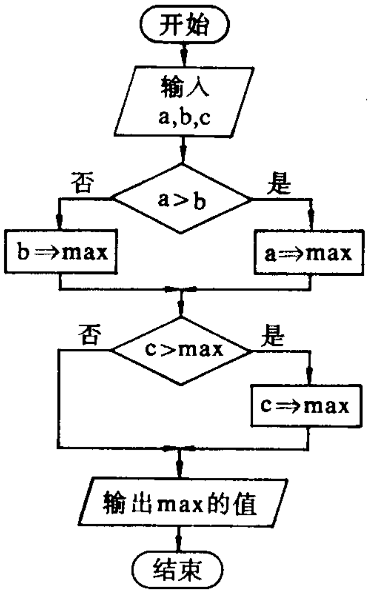


图 1.6

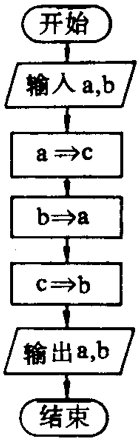


图 1.7

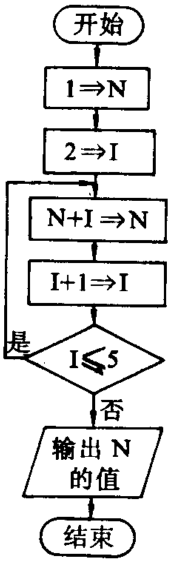


图 1.8

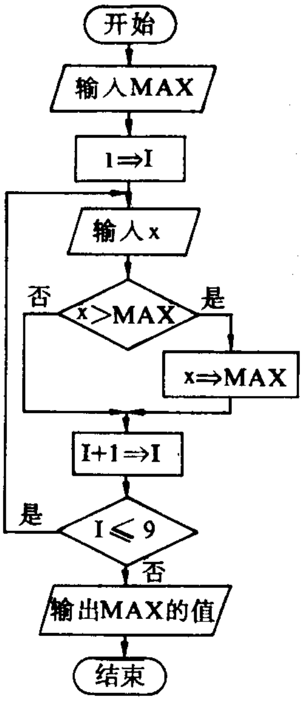


图 1.9

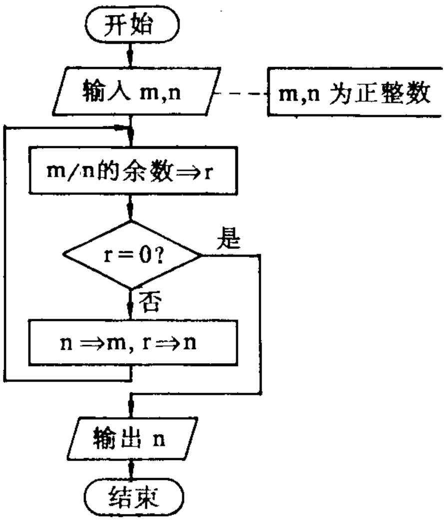


图 1.10

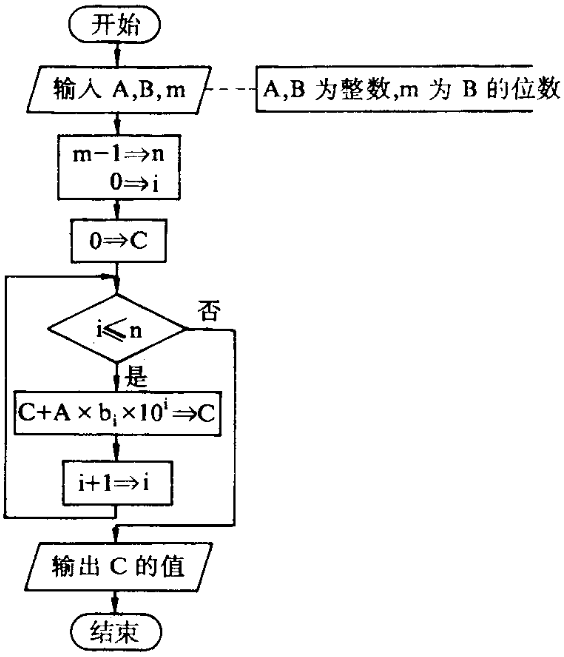


图 1.11

$$B=b_2b_1b_0=5\times10^2+1\times10^1+3\times10^0$$

图 1.11 中的循环部分共执行 3 次。

1.2.3 三种基本结构

用流程图表示算法方便直观，画图方法比较自由灵活，但是它存在着一个致命的弊病：

由于流程图要用带箭头的流程线指明执行的先后顺序，对流程线的使用也无任何限制，即允许从某一处用流程线将流程转向流程图中的任何一个框的入口处，看起来方便，但随心所欲不加任何限制地在流程图中用流程线使流程无规则地转来转去，由于问题的规模和复杂程度的增加，将使流程图变得象乱麻一样，使人难以理解。或者说，这种算法结构没有规律、清晰度差，使人难以阅读。图 1.12 是这种算法结构的示意图，其中每一根横线代表一个框的功能。由于这种算法结构象“面条”一样(象一碗面条一样互相缠绕在一起，难以找到头和尾)，故被称之为 BS(Bowl of Spaghetti)型，也有的称其为耗子窝(rat's nest)型。

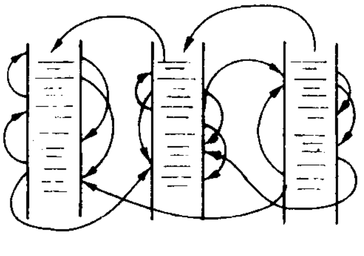


图 1.12

显然，这种 BS 型结构不利于阅读和交流，也不便于修改，从而使算法的可靠性和可维护性难以保证。为了提高算法的质量，必须限制箭头的滥用，使算法结构规范化。

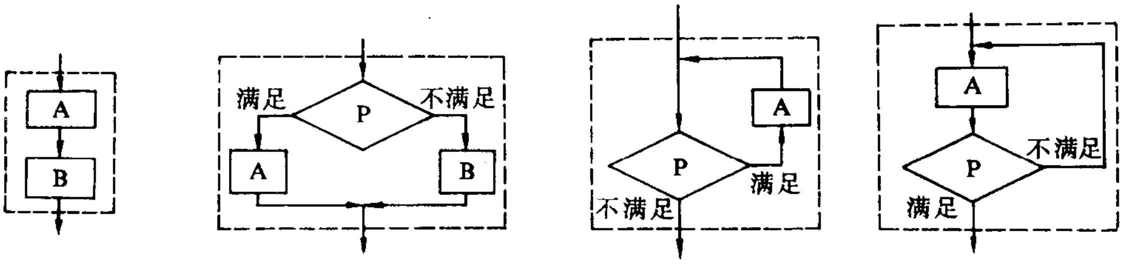
1966 年 Bohra 和 Jacopini 提出了结构化算法的三种基本结构单元：

(1) 顺序结构；

(2) 选取结构(或称选择结构)；

(3) 循环结构(或称重复结构)。

这三种基本结构的流程图，如图 1.13 所示。



(a) 顺序结构　(b) 选取结构　(c) 循环结构(当型)　(d) 循环结构(直到型)

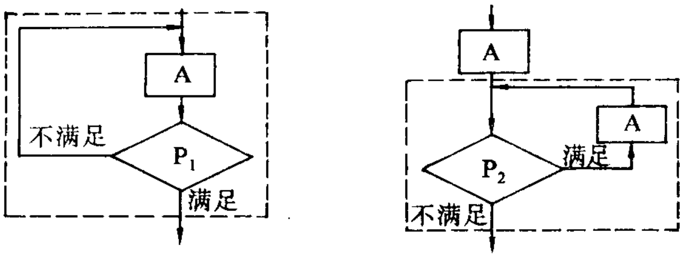
图 1.13

图 1.13(a)表示顺序结构，要求算法应按书写的顺序执行，即先执行 A 操作，再执行 B 操作。

图 1.13(b)表示二路分支的选取结构，要求执行过程中根据条件 P 是否满足选择执行 A 操作或 B 操作。

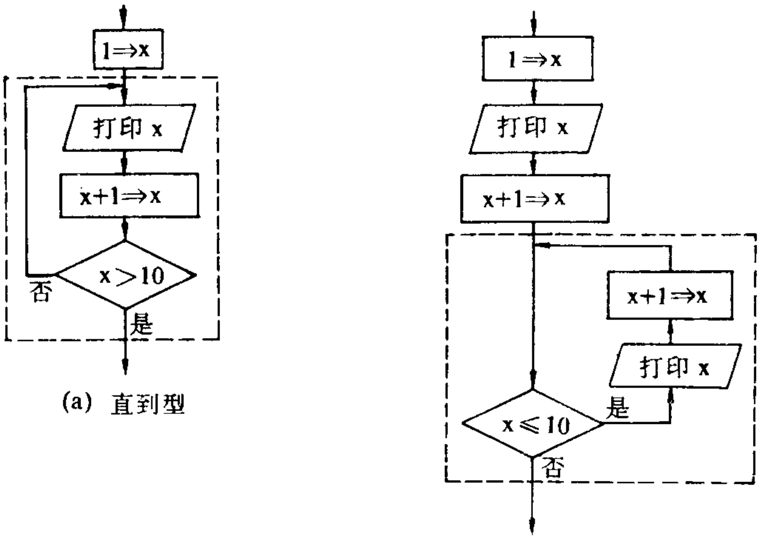
图 1.13(c)与(d)是循环结构的两种形式。图 1.13(c)称为“当型”(或 WHILE 型)循环，它要示当条件 P 满足时执行 A 操作。显然，当一开始条件 P 就不满足时，A 操作不会被执行。图 1.13(d)称为“直到型”(或 UNTIL 型)循环，它要求执行操作 A，直到条件 P 满足为止。显然，不管条件 P 满足还是不满足，A 操作至少要执行一次。所以当要把一个如图 1.14(a)所示的直到型循环结构转换成当型循环结构时，应先在循环结构之外复制一个 A 框，然后将条件 P_1 改为 P_2，P_2 是 P_1 的“反条件”，即 $P_2=\overline{P}_1$，如图 1.14(b)所示。例如，图 1.15(a)的直到型循环结构可转换为图 1.15(b)所示的当型循环结构。请注意，图 1.15(a)中的判断

条件为“x>10”，而图 1.15(b)中的判断条件为“x≤10”，二者互为反条件。同时应注意菱形判断框两侧的“是”和“否”的位置恰好相反，直到型循环(图 1.15(a))是条件满足时不再重复执行当中的两个框(循环体部分)，而当型循环(图 1.15(b))则是在条件满足时执行该两个框。



(a) 直到型　　(b) 当型

图 1.14



(a) 直到型　　(b) 当型

图 1.15

从图 1.13 可以看出，这三种基本结构具有如下特点：

(1) 只有一个入口。

(2) 只有一个出口。

(3) 每一个框内的功能都有机会被执行。即对每一个框来说，应当有一条从入口到出口的路径通过它。图 1.16 就不具有这样的性质。

(4) 不包含死循环(即无休止的循环)。图 1.17 就不具有这样的性质。

已经证明，任何复杂的问题，都可以用以上三种基本结构顺序地构成。当然，除此三种结构外，算法还可以有其它结构。但是将算法结构限制为只使用上述三种基本结构后，可以使算法的各基本单元(即基本结构)之间形成顺序执行关系。每一个基本结构具有单入口单出口的性质，使不同基本结构之间的关系简单，互相依赖性较少，它们是互相独立的“元部件”，

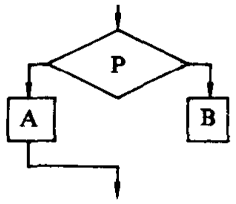


图 1.16

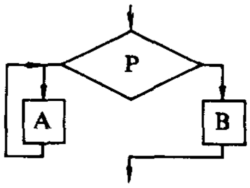


图 1.17

人们能够像搭积木一样将它们搭成各种不同的大结构。整个算法看起来就像项链上的一串珍珠一样由各基本结构顺序组成，结构清晰，从而可以使算法的质量显著提高。符合这一原则的流程图称为结构化的流程图。如图 1.18 就是一个结构化的流程图。因为它可以看作由 J,I,H 三个基本结构串接而成。而 I 是由 K,L 与 P_1 组合而成的选取型结构；K 是由 M,N 与 P_2 组成的选取型结构；M 与 N 都是基本结构。L 也是由基本结构组合而成的。

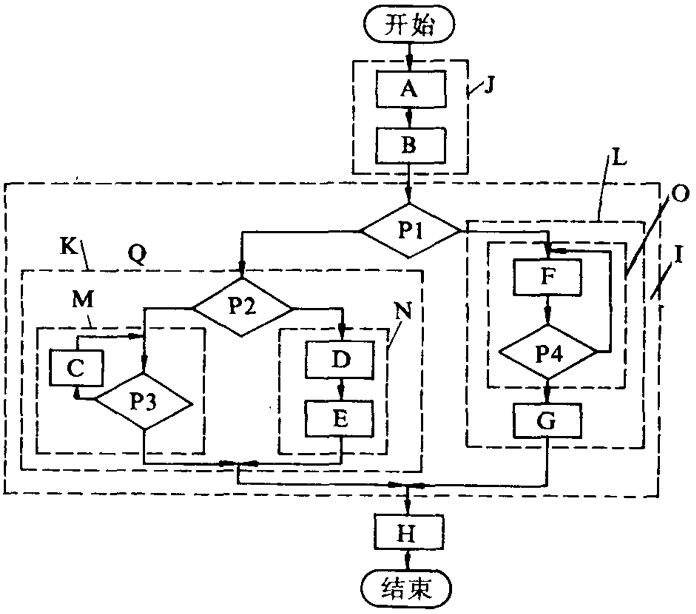
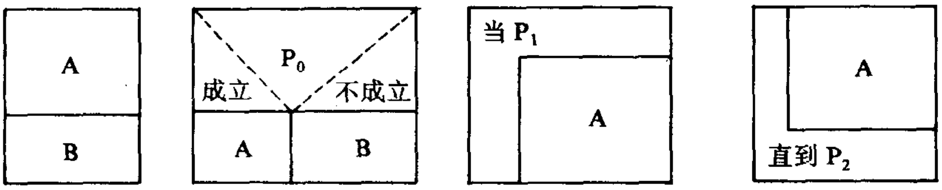


图 1.18

1.2.4 N-S 结构流程图

传统的流程图允许箭头随意转移，用这种流程图不能保证流程图是结构化的，而且这种流程图占篇幅比较大，作图工作量也较大。

1973 年美国学者 I. Nassi 和 B. Shneiderman 提出了一种新的流程图工具。由于他们二人的名字是以 N 和 S 开头的，因此，这种流程图称为 N-S 图。N-S 图完全去掉了在算法描述中引起麻烦的带箭头的流程线，全部算法写在一个大的矩形框内。规定了一些图形元素，以表示三种基本结构，每个元素用不同的框来表示。或者说，一个算法框是由一些代表基本结构的基本框象堆积木一样构造而成的。由于 N-S 图像一个多层的盒子，也称盒图(box diagram)。图 1.19 示出了它的三种基本结构图。



(a) (b) (c) (d)

图 1.19

1. 顺序结构：如图 1.19(a)所示，其中 A 和 B 分别代表一个基本操作(例如加、减、乘、除运算，打印、赋值、……)。两个或多个矩阵框顺序组成一个顺序结构。

2. 选取结构：如图 1.19(b)所示。它表示当 P_0 条件成立时执行 A 操作，P_0 条件不成立时执行 B 操作。图 1.20 是打印出 a 的绝对值的算法描述。

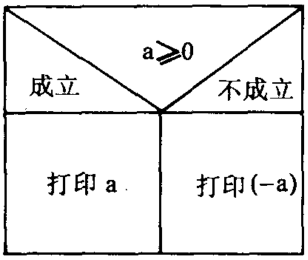


图 1.20

3. 循环结构：

(1) 当型循环：用图 1.19(c)的形式表示。当型表示当 P_1 条件满足时执行 A 操作，然后再返回判断 P_1 条件是否满足，如满足再执行 A，……，如此重复下去，直到 P_1 条件不满足为止。

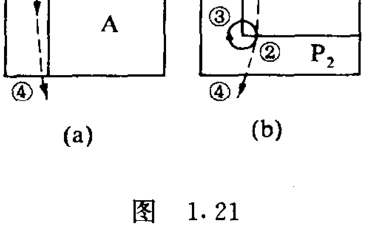
(2) 直到型循环：见图 1.19(d)。它表示，先执行 A 操作，再判断 P_2 条件是否满足，如不满足则返回再执行 A 操作，如满足则不再继续执行循环。

图 1.19(c)(d)中，"当 P_1"和"直到 P_2"，也可以写成"while P_1"和"until P_2"。有时也可以省写"当"、"直到"、"while"、"until"，直接写"P"。从图案的形状即可知道它是"当型"还是"直到型"结构。前已述及，这两种结构是可以互相转换的。

当型循环与直到型循环的执行过程略有差异。由图 1.21 可以形象地看出：当型循环中，流程从上方进入本结构，先遇到 P_1(图中以①表示)，判断 P_1 条件，再往下执行 A 操作(图中以②表示)，然后再返回判断 P_1(以③表示)，……，到某次判断 P_1 不再满足，就脱离本结构，执行下一个结构(图中以④表示)。

从图 1.21(b)可以看到：直到型循环结构则先执行 A(图中①)，然后判断 P_2 条件(图中②)，如 P_2 不满足再返回执行 A(图中③)，周而复始，直到 P_2 成立，则脱离本结构(图中④)。

在一种基本结构中还可以包括另一个基本结构。如图 1.22 中，包含两个框，第二个框是



(a) (b)

图 1.21

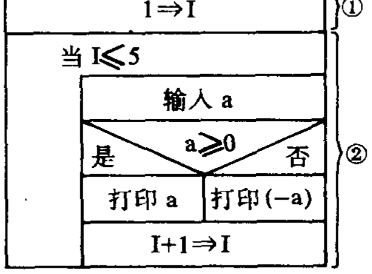


图 1.22

一个当型循环。在该当型循环中，被重复执行的部分（循环体）又包括了三个框，其中有一个选取结构。这个算法的目的是：先后输入 5 个数，打印出每一个数的绝对值。I 的作用是用来累计“次数”，处理完 5 个数之后，I＝6，不再执行循不。

可以将图 1.22 中的选取结构用一个简单的矩形代表，见图 1.23。实际上，任何一个基本结构都可以用一个简单的框代表：图 1.23 是一个“粗流程图”，图 1-22 是“细流程图”。开始设计时可以粗一些（如图 1.23），在详细设计时，再将“打印 a 的绝对值”细化为图 1.22 中的选取结构。图 1.23 中当型循环中的三个框又可组成一个顺序结构。因此，它就是图 1.19 (c) 所示的当型循环形式。

用 N-S 图表示例 1.11 到例 1.16 中的算法，见图 1.24～图 1.30。

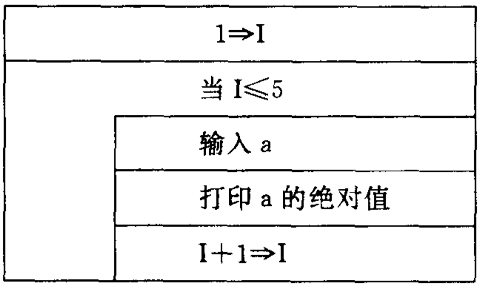


图 1.23

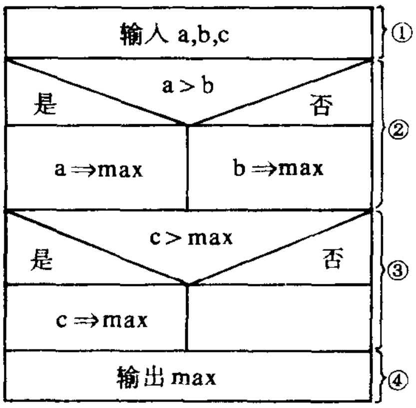


图 1.24

例 1.2.7 图 1.24 表示的是从三个数中找最大数的算法。图 1.24 的 N-S 图由 4 个框组成，第二、四两个框是选取结构。可以明显地看出，这个算法是结构化的，它是由基本结构组成的。执行此算法时，从上到下逐个执行各个基本结构。

例 1.2.8 表示交换两个变量值的算法的 N-S 图，如图 1.25 所示。

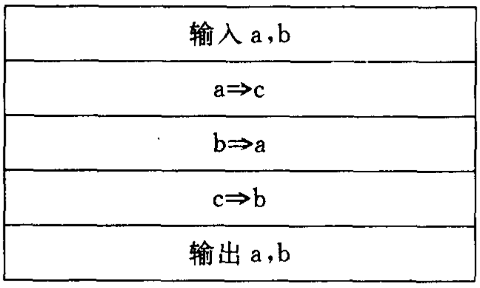


图 1.25

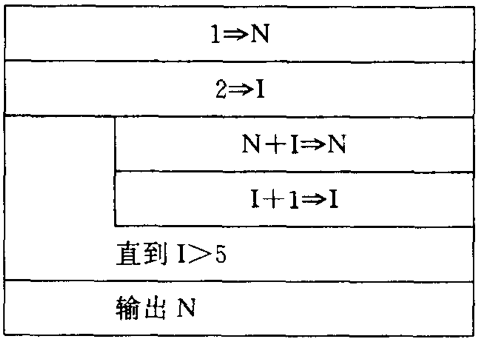


图 1.26

例 1.2.9 表示 $\sum_{n=1}^{5} n$ 算法的 N-S 图，见图 1.26。

请将图 1.26 与图 1.8 对比。图 1.26 中用直到型循环形式，结束循环的条件是 I>5，而图 1.8 中表示继续执行循环的条件是 I≤5，当 I≤5 条件的判断结果为否（即 I>5）时，就不再执行循环了。图 1.26 和图 1.8 表示的是同一件事，只是表示方法不同而已。也可以将图 1.8 中的菱形框内的条件改为 I>5，同时将两个出口的“是”和“否”颠倒一下，作用是一

样的。

例 1.2.10 输入 10 个数，打印出其中最大的数。其 N-S 图见图 1.27。

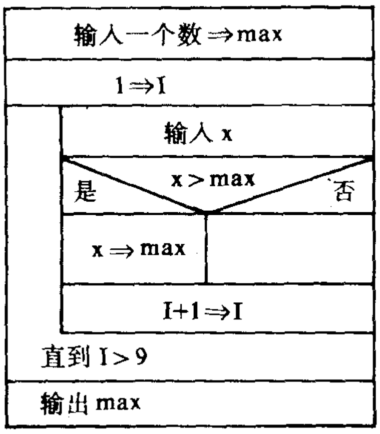
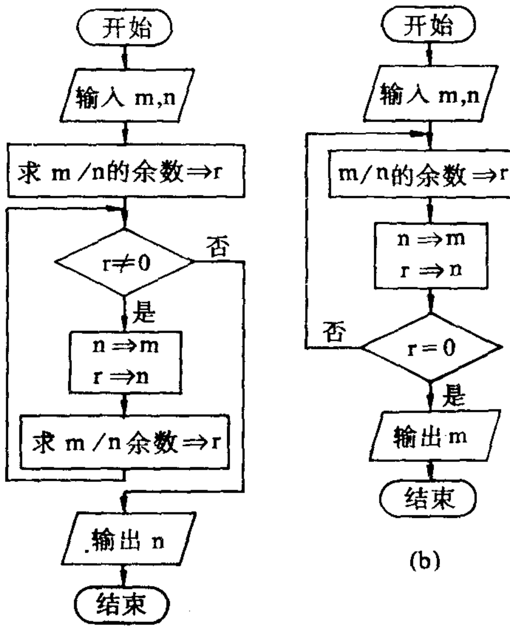
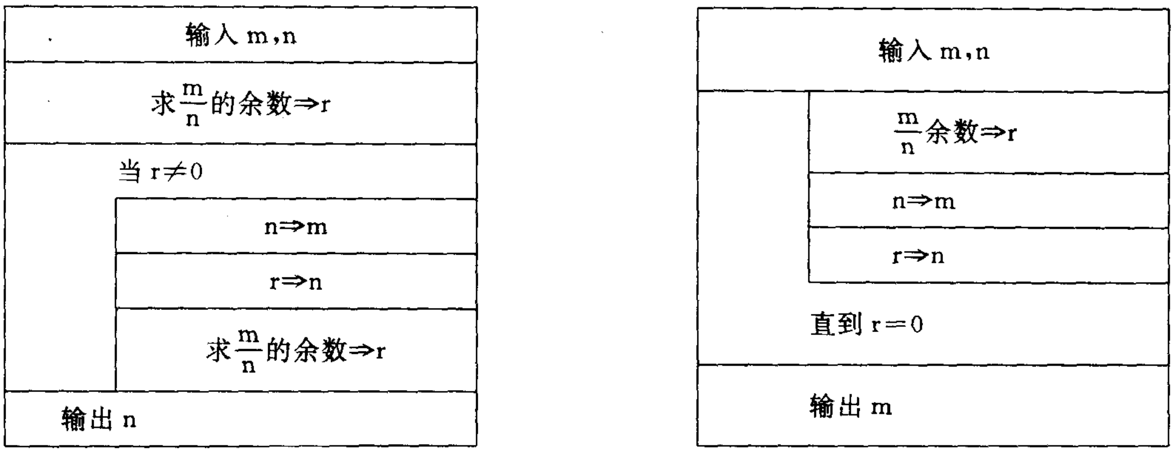


图 1.27



(a) (b)

图 1.28



(a) (b)

图 1.29

例 1.2.11 求 m,n 的最大公约数。

先看一下图 1.10，其中的循环不是标准的当型或直到型循环，可先将其改画成图 1.28(a)。

图 1.28(a)是一个当型循环。可以将它改画成 N-S 图形式，见图 1.29(a)，也可画成直到型循环，见图 1.28(b)和图 1.29(b)。

例 1.2.12 求两个正整数 A 和 B 的乘积。其 N-S 图见图 1.30。

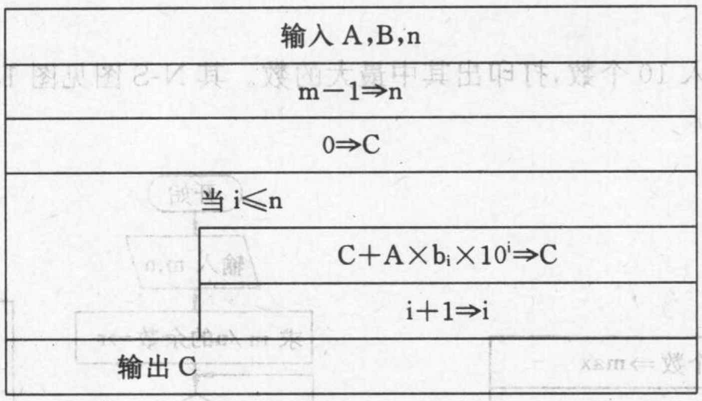


图　1.30

请读者将图 1.24～图 1.30 与图 1.6～图 1.11 对照，比较这两种方法。显然，用 N-S 结构流程图的优点：(1)能够保证流程图是由三种基本结构顺序组成的，它必然是结构化的，因此完全可以不要带箭头的流程线。(2)画 N-S 图比画传统流程图容易而且节省篇幅。(3)N-S 图阅读起来直观、明确，容易理解。因此我们推荐用 N-S 图来表示算法。使用 N-S 图可以强迫自己养成写结构化算法的良好习惯。

1.3　用“逐步细化”方法进行算法设计

对于一个简单的解题任务，可以直接写出详细算法，如前面我们所做的那样。但是对于一个复杂的问题，要一下子直接写出其每一个操作步骤是困难的。这时可以采用“自顶向下，逐步细化”的方法，从粗到细地逐步具体化。例如，写一篇文章，往往并不是一下子想好了全部用词，而是先想好文章的题目和要求，再想它应该分成哪几个部分，每个部分分哪几段，最后再具体组织词句。这种从全局到局部，从抽象到具体，一步一步细化的方法是处理复杂问题的有效方法。

在算法设计中，“抽象”是一种重要的方法。由于在求解复杂问题时，人的智力往往不可能一下子就能达到问题的每一个细节，因此，在开始阶段，可以用“抽象的描述”来概括一些具体的细节。也就是说，抽象的目的是为了“隐藏”细节，以便更好地面向问题、分析问题。

在拿到一个任务之后，首先要分析该问题的要求。先解决需要“做什么”的问题(而不急于解决“怎么做”的问题)，可以用自然语言或是数学公式来描述“做什么”，这就是抽象算法。然后，逐步进入“实现级”，即不断将这些抽象算法具体化，进入较低级的抽象级，逐步解决“如何做”的问题，直到最后把全部算法都描述成可由计算机实现的基本操作为止。这种基于抽象的算法设计方法就称为“逐步细化”(step-nise refinement)，或称“逐步求精”。简单地说，逐步细化过程是一个由较高抽象级逐步转向较低抽象级的过程。每一个抽象级上的问题都应当是人们的智力所能及的问题(即“子问题”)。它要求人们在某一抽象级上集中精力分析与之有关的问题，而避免涉及与之暂时无关的低层细节。把这一层问题解决好后，才进入下一层，把每个子问题再精细化。可以采用分割(将一个问题划分为互相独立的一些子问题)、分析(将问题严格分类，对不同的问题采用不同的解法)、递推(重复使用某种算法，使问题不断达到最终解决)、以及穷举等策略使问题精细化。

为了清楚地表示算法设计过程中每一级的算法，将采用 N-S 图来表示算法。它是一种

表示算法的较好的工具。它独立于任何计算机或计算机语言，但是又能很方便地转换成任一种计算机语言程序。在设计算法时，可以完全不考虑使用什么计算机和计算机语言。待算法设计完成后，再根据需要转换成特定的计算机语言程序。因此，用 N-S 图或伪代码等表示算法可以认为是一种“抽象的程序”，根据“抽象的程序”设计出“具体的程序”。应当掌握这种设计方法。

下面举几个简单的例子说明如何用“逐步细化”方法进行算法设计。

例 1.3.1 找出三个数中的最大数。

在上一节中，已直接写出算法，见例 1.1.1 和例 1.2.1 以及例 1.2.7。现在再利用逐步细化方法进行算法设计，帮助读者了解如何运用这种方法。

首先在“做什么”一级上把问题抽象为图 1.31(a)所示的三个子问题 S1，S2，S3，再分别对三个子问题细化。

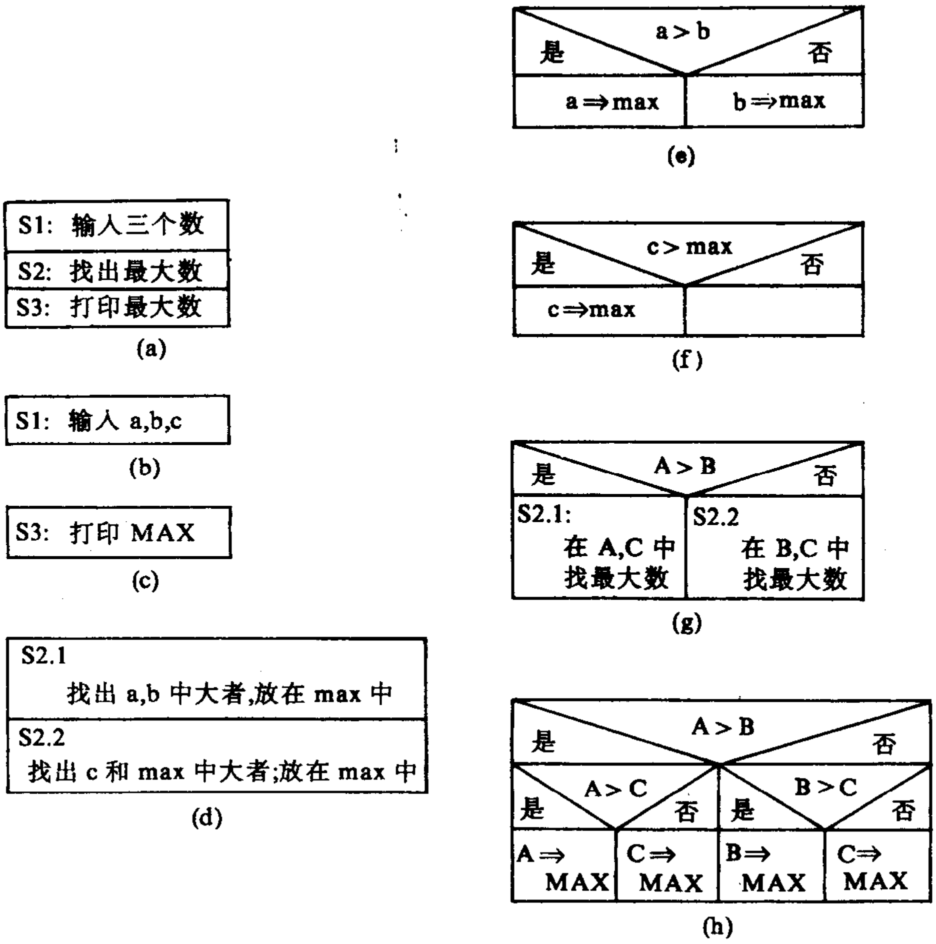


图 1.31

若用 a，b，c 分别代表这三个数，并用 MAX 存放其中最大的数，则可以把 S1 与 S3 描述为图 1.31(b)与(c)所示的方法。这两个算法已成为计算机可以实现的基本操作。对 S2 来说，可以用分割技术，将 S2 分成 S2.1 和 S2.2 两个独立的子任务。再分别对 S2.1 和 S2.2 细化，可以采取分析技术，将 S2.1 细化为图 1.31(e)，S2.2 细化为图 1.31(f)。请读者将图

1.31 中的(b)、(e)、(f)、(c)顺序连起来画出完整的流程图。不难看出，它就是图 1.24 所示的 N-S 流程图。

也可以采取另一个方法对图 1.31(a)中的 S2 细化。可以用分析技术，以 A>B 是否成立，把问题细化为“在 A,C 中找最大数”或“在 B,C 中找最大数”两个子问题，如图 1.31(g)所示。再用分析技术可以得到图 1.31(h)所示的算法。这时 S2 已全部为计算机可以执行的基本操作描述了。请读者将图 1.31 中的(b),(h),(c)顺序画出完整的 N-S 结构流程图。

由本例可以看出：分析技术通常用选取结构来实现。

例 1.3.2 用筛法求 1 到 1000 之间的全部素数(质数)。

求素数表的方法很多，但是有一种方法与众不同。它就是“埃拉托色尼(Eratost-henes)筛法”。埃拉托色尼是与欧几里德差不多同时代的希腊著名数学家，他求出了世界上第一张素数表。他的办法是：在一张纸上写上 1 到 1000 的全部整数，然后进行以下操作：

S1：挖掉 1；

S2：重复执行如下操作：用下一个未被挖掉的数 x_j 去除它后面的每一个未被挖掉的数 x_i，把能整除的数挖掉，直到这个数后面没有可以被挖掉的数为止。

S3：输出纸上所剩下的数。

执行这个算法后，纸被挖得象一个筛子，如图 1.32 所示。所以这种求素数的方法被称为筛法。

① 2 3 ④ 5 ⑥ 7 ⑧ ⑨ ⑩ 11 ⑫ 13 ⑭ ⑮ ⑯ 17 ⑱ 19 ⑳
㉑ ㉒ 23 ㉔ ㉕ ㉖ ㉗ ㉘ 29 ㉚ 31 ㉜ ㉝ ㉞ ㉟ ㊱ 37 ㊳ ㊴ ㊵
41 ㊷ 43 ㊹ ㊺ ㊻ 47 ㊽ ㊾ ㊿ ……

图 1.32

下面介绍如何用逐步细化方法导出这一算法。

首先在“做什么”一级把问题分割为三个子问题。

S1：把自然数 1,2,…,N 赋给变量 $x_1,x_2,\cdots,x_N$。

S2：把 1～N 中的非素数去掉。

S3：打印 1～N 中未被去掉的数，即素数。见图 1.33。

S1：把自然数 1,2,…,N 赋给 $x_1,\cdots,x_n$
S2：把 1～N 中非素数去掉
S3：打印 1～N 中未被去掉的数

图 1.33

$1\Rightarrow x_1$
$2\Rightarrow x_2$
⋮
$N\Rightarrow x_N$

图 1.34

下面对这三个子问题再细化：

1. 对 S1 细化：将 1～N 分别送给 x_1～x_N，可以用图 1.34 所示算法。但是这种算法太不经济。进一步分析后发现如下规律：

(1) 每个变量要接收的数，与其下标相同；

(2) i 依次增 1。

因此可以把 S1 重新描述为：i 送给 x_i，其中 i=1,2,…,N。

这一描述可细化为图 1.35 所示的算法。用循环结构来实现把 N 个数送到 N 个变量的过程。在进入循环结构前，要先确定 N，并给 i 以初值。

2. 对 S2 细化：对 S2 的功能进行分析，可以把它进一步分割为如下两个子问题：

S2.1：把 x_1 去掉；

S2.2：将 x_2 到 x_N 间的非素数去掉。

S2.1 已经是一个基本操作。下面仅对 S2.2 细化，可以得到图 1.36 所示的算法。

S1

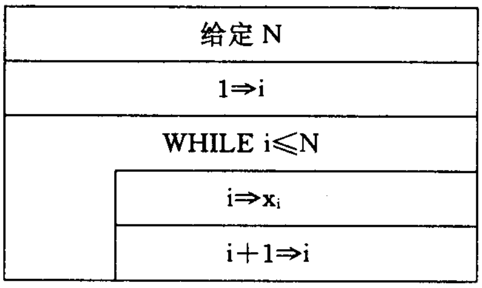


图 1.35

S2.2

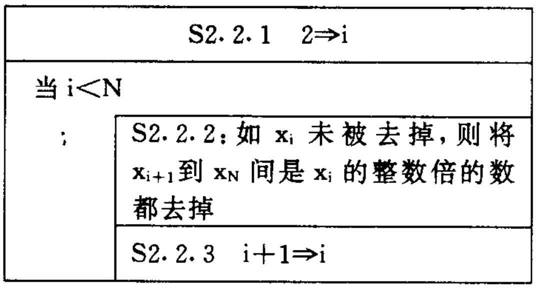


图 1.36

在 S2.2 中，S2.2.1 与 S2.2.3 已为基本操作，下面仅对 S2.2.2 细化，得到图 1.37 所示的算法，下面再对 S2.2.2 细化。S2.2.2 是要将 x_{i+1}到 x_N 间能被 x_i 整除的数都去掉，这就需要从 x_{i+1}开始到 x_N 为止一个一个判断它们是不是可以被 x_i 整除。由于对每一个数的判断操作都是相同的，所以我们可以使用循环结构来实现。见图 1.38，这时我们要设置另外一个变量 x_j 来代表 $x_{i+1}, x_{i+2}, \cdots, x_N$ 中的某一个数。在重复判断过程中，先让 j=i+1，然后，每判断一次，令 j+1⇒j。这样，开始时 $x_j=x_{i+1}$，当 j+1⇒j 后，x_j 就等于 $x_{i+2}, \cdots$，j 不断改变，直到 $x_j=x_n$ 为止。

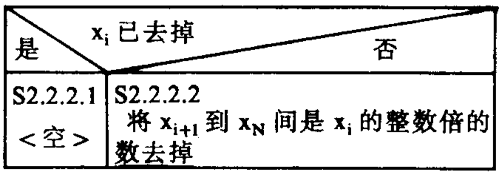


图 1.37

S2.2.2.2

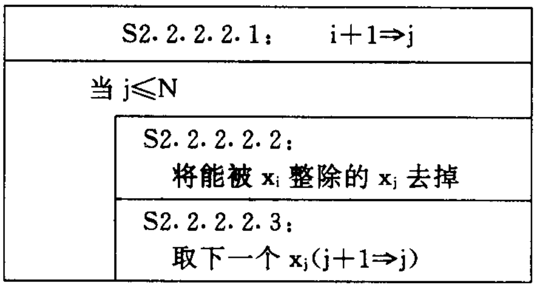


图 1.38

下面需要再对 S2.2.2.2.2 求精，解决如何将一个能被 x_i 整除的 x_j 去掉。检查前，首先要看一下 x_j 是否已被去掉，只有 x_j 是一个未被去掉的数时，才要决定是否要将其去掉，去掉 x_j 的条件是 x_j 能被 x_i 整除。于是我们可以得到图 1.39 所示的算法。到此为止，每一个操作都是基本操作了，细化过程也就结束了。下面只要把“将 x_j 去掉”改为计算机可执行的操作“0⇒x_j”，并把相应的命题“x_j 已被去掉”改为“$x_j=0$”，便可以得到图 1.40 所示的 S2 算法。其中如何判断“x_j 能被 x_i 整除”的问题，我们以后再作讨论。

3. 对 S3 细化：可以首先把 S3 理解为对 $x_1, x_2, \cdots, x_N$ 中的每个数依次判断，只要某 x_i 未被去掉，便把它打印出来，得图 1.41(a)所示的算法。对其中 S3.2 细化可得图 1.41(b)所

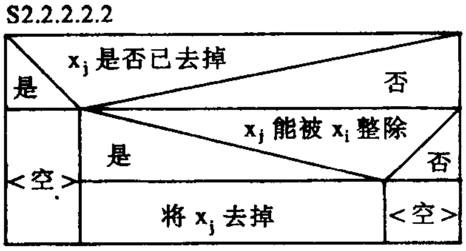


图 1.39

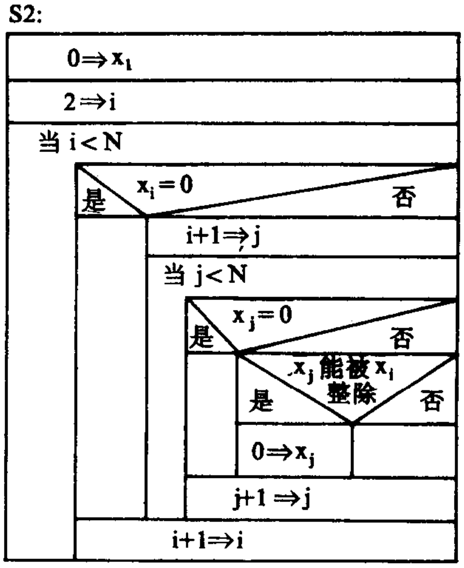


图 1.40

示算法。

将上述S1,S2,S3连接起来,便可得到图1.42所示的算法。

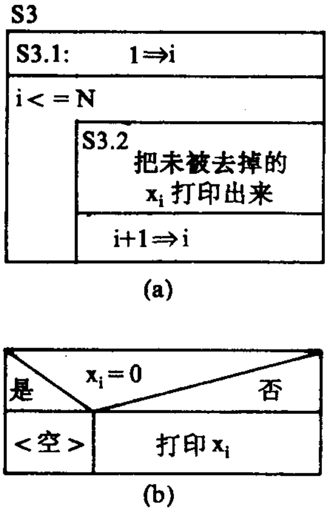


图 1.41

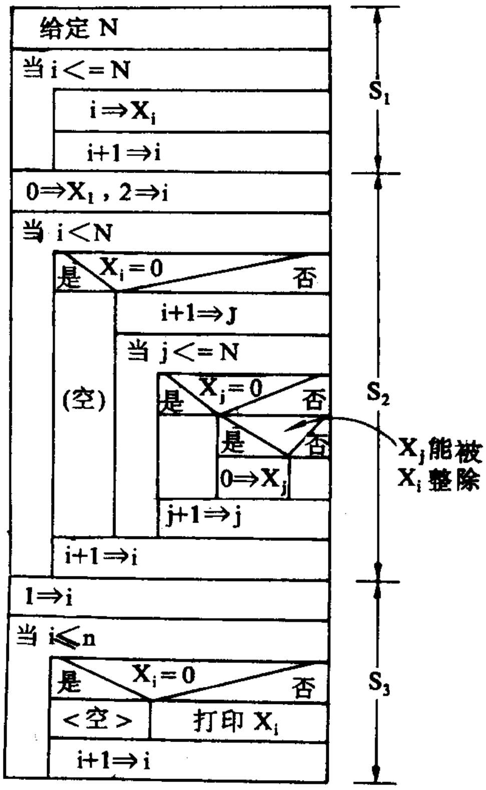


图 1.42

例1.3.2是一种从一个数据集当中找出符合某些条件的子集。用计算机求解这类问题

的基本方法是"穷举法",即对数据集中的每个元素逐一测试,把全部元素都测试过之后,才能找出问题的全部解。穷举法的基本算法结构是循环。或者说凡是可以归结为穷举问题的算法,可以考虑用循环结构来实现它。

循环结构的另一个重要的应用是实现递推过程。请见下例。

例 1.3.3 计算 $S=\frac{1}{1!}+\frac{1}{2!}+\cdots+\frac{1}{(i-1)!}+\frac{1}{i!}+\frac{1}{(i+1)!}+\frac{1}{20!}$

计算上式的值,一般是先算出各个阶乘的值,分别求其倒数,然后求和。这种算法所需的乘法运算次数较多。如果把上式稍作变换,可以得到下面的形式:

$$S=\frac{1}{1}\left(1+\frac{1}{2}\left(1+\frac{1}{3}\left(1+\cdots\left(1+\frac{1}{i-1}\left(1+\frac{1}{i}\left(1+\cdots\left(1+\frac{1}{20}\right)\right)\right)\right)\cdots\right)\right)\right)$$

这一表达式的特点:设某一圆括号内的值为 S_{i+1},则它的外面一对圆括号内的值为

$$S_i = 1 + S_{i+1}/i \qquad (1.1.1)$$

再外一层圆括号内的值为

$$S_{i-1} = 1 + S_i/(i-1) \qquad (1.1.2)$$

这样便可以由某一个内层的值,逐步递推出各外层的值,直到整个解。由于递推的算法本身是不改变的,所以有可能采用循环结构来实现它。按照这一思想得到的本题的算法如图 1.43 所示。这种算法设计技术称为递推。使用递推技术的关键是如何导出问题的递推形式。

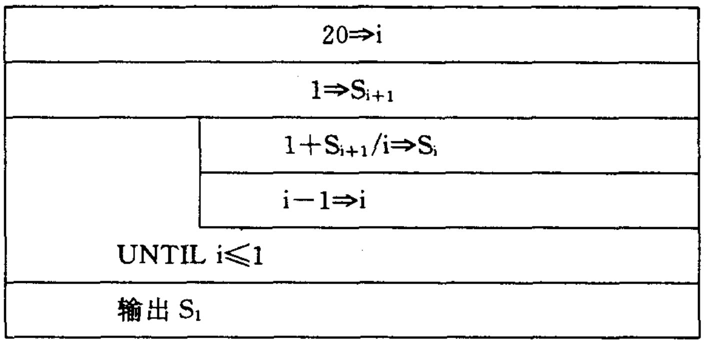


图 1.43

递推的一种重要形式是"倒推",即从已知的最后结果倒推出来未知的初始值,或者是说,从某个目标(即最后结果)出发,倒推出这个问题的初始情况。

例 1.3.4 小猴吃桃:有一天小猴子摘下了若干个桃子,当即吃掉了一半,还觉得不过瘾,又多吃了一个。第二天接着吃了剩下的桃子中的一半,仍不过瘾,又多吃了一个。以后每天都是吃尚存桃子的一半零一个。到第 10 天早上小猴子再去吃桃子时,看到只剩下一个桃子了。问小猴子第一天共摘下了多少个桃子。

首先分析一下用代数方法如何求解此题。

设小猴子共摘了 x 个桃子,根据题意可以有:

第 1 天吃掉 $\frac{x}{2}+1$(个),剩下 $\frac{x}{2}-1=\frac{x-2}{2}$(个);

第 2 天吃掉 $\frac{1}{2}\left(\frac{x-2}{2}\right)+1$(个),剩下 $\frac{x-2}{2}-\left[\frac{1}{2}\left(\frac{x-2}{2}\right)+1\right]=\frac{x-2}{2}-\frac{x-2+2^2}{2^2}=\frac{x-2-2^2}{2^2}$(个);

⋮

第 i 天剩下$\frac{x-2-2^2-\cdots-2^i}{2^i}$(个)；

⋮

第 9 天剩下$\frac{x-2-2^2-\cdots-2^9}{2^9}$(个)，即第 10 天猴子看到的那一个桃子。故有

$$\frac{x-2-2^2-\cdots-2^9}{2^9}=1$$

解此方程得

$$x=2^9+2^9+2^8+2^7+2^6+\cdots+2^2+2=1534(个)$$

根据此式可以编写出解题程序，但程序将执行一个较复杂的计算。如果题目给的不是到第 10 天才剩一个，而是到第 100 天才剩一个，则有 $x=2^{99}+2^{99}+2^{98}+\cdots+2$，计算更为复杂。若用倒推（"倒推"）法计算就简单多了。其基本思想是，从最后一天起逐次推出前一天的桃子数，直到推出第一天吃桃子前的全部桃子数为止。具体方法如下：

根据题意，猴子某天吃桃子前的桃子数是前一天吃桃子之前的桃子数的一半减 1，即

$$PEACH_n=\frac{1}{2}PEACH_{n-1}-1$$

亦即 $PEACH_{n-1}=(PEACH_n+1)\times 2$

这样就可以从第 n 天的桃子数倒推出第(n−1)天的桃子数（都是指该天未吃桃子前的数），并一直可以递推求出最初的值。反复地递推过程可以用循环结构实现，如图 1.44 所示。

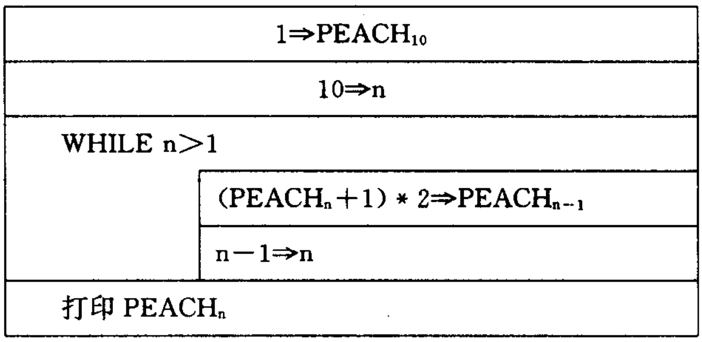


图 1.44

倒推法是一种常用的算法设计策略。如果一个问题有唯一解，并且它们的运算是可逆的（即对每个运算来说，输入与输出有一对一的关系），就有可能用倒推法对它求解。本例中，小猴子摘下的桃子数显然是一定的——有唯一解。按照小猴子的吃法，每天剩下的桃子数与它吃之前的桃子数必然是一对一的，即由前一天的桃子数只能得到其后一天的唯一的桃子数，反之，由后一天的桃子数只能推出前一天的唯一的桃子数。

一般来说，递推是指从前面一些量可以依次推出后面一些量的算法。它并不一定要求采用循环算法。但是，从前面的几个例子可以看出，当能找到用固定算法的递推形式时，用循环结构是很经济的。递推常常表现为迭代形式，即由一个变量的原值推出它的新值，或者说，不断地用一个新值去代替变量的原值。原值与新值之间存在一定的关系，用迭代公式来表示。

迭代（以及递推）和穷举是循环型算法的两个主要应用形式。

在本节中只介绍了算法设计的一些简单例子，在以后各章中将结合程序介绍算法设计。

1.4 程序设计语言

1.4.1 概述

前已述及，电子计算机是实现算法的有效工具。为了让计算机执行人们设计的算法，必须把算法最后描述成计算机能够理解和执行的形式。程序设计语言是人和计算机交流信息的工具，一个算法只有最后用程序设计语言来描述才能被机器接收、理解并执行。从这个意义上讲，程序就是用程序设计语言描述的算法。

请读者弄清楚"语言"与"程序"的区别。用打算盘的例子来说明这个问题。如果想计算978＋24，将被加数978放好，要加上24，可以用以下口诀："四去六进一，二去八进一，一去九进一"，得1006。这三个口诀称为三条"指令"，每条指令规定了一个基本操作。这三句口诀(即指令)组成一个求978＋24的"程序"。而算盘的全部口诀之和就是"算盘语言。"这就是说"语言"是全部指令的总和。要实现一个算法，可从该"语言"中选择所需要的指令组成程序。所以说程序是完成特定任务的一组指令序列。

1.4.2 程序设计语言的发展

随着计算机技术的发展，程序设计语言也在发展，并经历以下几个阶段：

第一代计算机语言——机器语言。是最早产生和使用的计算机语言，在40年代和50年代初期，人们只能使用这种语言去操作计算机。

第二代计算机语言——汇编语言。出现于50年代中期。使用它比使用机器语言更方便。

第三代计算机语言——算法语言。出现于50年代中后期。它接近自然语言与数学语言，容易掌握。

第四代计算机语言——非过程化的语言，使用更方便。人们只要把问题描述成"做什么"，机器接收后便能求解。

对这几种语言分述如下：

1. 机器语言(machine language)：如前所述，电子计算机是由电子元件(例如二极管、三极管)组成的。它们的作用好似"开关"，只有两种稳定的工作状态：导通和截止，分别以1和0代表。

假如用8个这样的"电子开关"排成一行，它们的"导通和截止"状态就可以用8位"0""1"数字来表示，例如0 1 0 0 1 0 0 0，表示第二个和第五个开关导通，其余的为截止。可以看到，由电子元件组成的电子线路的工作状态用二进制数来表示是很合适的。

机器语言中的指令是一个由0或1码组成的序列。有的计算机用一个字节(即8位)的二进制数信息来表示一个指令，有的则以2个字节或4个字节(即16位或32位)的二进制数码表示一个指令。例如，某一种型号的计算机以8位二进制信息10000000表示一次加法A＋B⇒A：

A＋B⇒A

以10010000表示一次减法A－B⇒A：

A－B⇒A

以 01110000 表示传送操作 A⇒B：

[A]⇒[B]

有人可能会问：为什么计算机能接收 10000000 就会执行加法操作、接收 10010000 就会执行减法操作呢？这是由计算机的线路来保证的。在设计一种型号的计算机时，先确定好每一条指令与线路应产生的操作之间的对应关系。这如同打电话一样，在安装电话线路时就已确定好哪一个号码对应哪一个用户。以后有人拨某一电话号码时，相应用户的电话铃就会响，而不会引起别的用户电话铃响。

机器语言中的每一条指令，代表了机器可执行的一个基本操作。不同的计算机系统线路逻辑是不同的。因此，对不同的计算机即使是执行同一个操作，其指令也是不同的。例如对某一型号计算机，执行加法的指令是 10000000，而另一型号计算机，就可能不是 10000000 而是 10100000。换句话说，每种型号的计算机都有自己专用的指令系统(instruelion set)。所谓指令系统就是指某一型号的计算机所能执行的全部操作指令的集合。每种型号的计算机只能理解和执行自己的指令系统中的指令。例如，长城 0520 计算机的指令在紫金 I 计算机是不适用的。把每一型号计算机的指令系统称该型号计算机的机器语言。它是面向具体机器的。

使用机器语言描述的算法，机器能直接识别和执行，执行效率也很高。但这种由 0 和 1 代码组成的二进制数信息含义不直观，难记难辨认，要写出这样的机器语言程序，只能靠“解密码”一样的艰巨的手工劳动，而且不便于交流和查错，只有熟练的计算机专门人员才能掌握它，这就使计算机的应用面局限于少数人的范围。

2. 汇编语言(assembly language)：50 年代后期，人们创造了汇编语言。它用一些简单的“助记忆符号”来代替二进制数代码，所以又叫做“符号语言”。例如，上一节介绍过的三条指令可以改用专门助记忆符号表示如下：

ADD A,B(执行加法 A+B⇒A)
SUB A,B(执行减法 A−B⇒A)
LD A,B(传递 A⇒B)

所谓助记忆符号是指人们约定的某些容易为人们记忆和理解的符号。如上面的 ADD 在英文中就是“加”，SUB 就是 Subtract(减)的缩写，LD 是 Load(装入)的缩写。显然，用“ADD A,B”要比用 10000000 形式的指令容易理解和记忆。

汇编语言的指令和机器语言中的指令是一一对应的，即一条机器语言指令要由一条汇编语言代替。如果执行某一组计算需要用 100 条机器指令去实现，则改用汇编语言也需要用 100 条(后来发展了宏汇编指令，可以用一条汇编指令代替若干条机器指令。但是，使用汇编语言编程序自然是不方便的)。

但是，计算机并不能直接识别和执行汇编语言的指令，需要将它们逐条译成二进制形式的机器指令，才能被计算机识别和执行。这个翻译转换的过程称为“代真”。代真后得到的机器语言程序称为“目标程序”(Object Program)。代真以前的汇编语言程序称为“源程序”(Source Program)。

最初，代真工作是由人工逐条进行的。后来人们发现，汇编语言指令和机器语言指令是一一对应的，这种代真实际上是一种机械地查表转换工作，这种工作完全可以由机器用一种

专用的翻译程序来完成。这种把汇编语言程序代真为机器语言程序称为“汇编程序”。这个代真的过程称为“汇编”。

汇编语言程序基本上保留了机器语言程序的优点，占存储容量小，执行速度快，提高了程序的易读性和程序设计的效率，但是汇编语言和机器语言一样都是面向机器的语言，它们被称为“低级语言”。对不同的计算机，汇编语言也是不同的。用汇编语言编程序，也必须先要了解计算机的内部结构，进行程序设计时要人工进行存储单元的分配，这对程序编写人员的要求是比较高的。编出的程序在不同机器之间的可移性差。在计算机日新月异的情况下，人们希望减少编写程序的劳动，提高程序的通用性，对计算机语言提出了更高的要求。

3. 高级语言(high level language)：机器语言和汇编语言都是面向机器的，使用不便，难学，难记，难写，难检查，使计算机推广应用受到很大的限制。人们希望用接近自然语言的工具来表示算法。如用“PRINT A”表示“打印 A 的值”，用“LET X=A+B”表示“使 X 的值等于 A 与 B 之和”，用“IF X>0 THEN PRINT A+B ELSE PRINT A-B”表示“假如 X>0 则打印 A+B 之值，否则打印 A-B 之值”。

这种表示方法很接近自然语言和数学语言，人们容易理解，而且用变量名代替了存储单元的地址，不必由人们直接分配地址。例如用 True BASIC 表示求 a+|b|的算法可以写成

```
INPUT a, b
IF b>=0 THEN LET c=a+b ELSE LET c=a-b
PRINT c
```

用当型循环结构求两个数的最大公约数的算法(见图 1.27(b))可以描述为

```
INPUT m, n
LET r=m/n
DO WHILE r<>0
    LET m=n
    LET n=r
    LET r=m/n
LOOP
PRINT n
END
```

而求两个数的最大公约数的直到型循环结构(见图 1.27(a))可以描述为

```
INPUT m,n
DO
  LET r=m/n
  LET m=n
  LET n=r
LOOP UNTIL r=0
PRINT m
END
```

上面的 PRINT 是让显示器显示结果或让打印机打印结果。a,b,c,m,n,r 是用以代表

存储单元地址的变量名。显然，这种表示对于略懂英语的人来说是不难理解的。高级语言不再依赖于具体的计算机，即用这种语言写的程序对各种型号的计算机基本上都可适用。也就是说，程序在不同的计算机之间具有可移植性，在一个机器上运行的程序可以不加修改（或只作小的修改）就可以拿到另一机器上运行。

高级语言不再是面向机器了，而是面向解题过程的。就是说，程序员不必再考虑机器内部构造和不同机器的特点，可以集中力量考虑解题的算法，写出解题的过程。目前用高级语言编写程序时，大都需要写出其执行过程，即用高级语言描述算法，这种高级语言也被称为"算法语言"(algorithmic langunge)或过程化语言。高级语言的产生是计算机发展史上一个"惊人的成就"，它扫除了计算机推广普及中的障碍，使广大计算机工作者（一般用户）能够进入计算机领域。人们根据不同的需要创造了不同的计算机高级语言。目前国内外使用较多的高级语言有一百多种，其中应用最广泛的有：

BASIC——一种应用广泛的高级语言，易学好用。

FORTRAN——世界上最早出现的高级语言，适用于数值计算。有 FORTRAN Ⅳ和 FORTRAN 77，后者可适用于结构化程序设计。

COBOL——适用于事务管理领域。

PASCAL——一种结构化语言。

ALGOL——适用于数值计算，有 ALGOL 60 和 ALGOL 68，后者功能很强。

PL/1——一种大型的语言，兼有 ALGOL，FORTRAN 和 COBOL 的特点，功能很强。

ADA——一种工程化的大型语言，现多用于国防领域。

True BASIC——一种基于 BASIC 并吸收了 FORTRAN 及 PASCAL 的优点的新型程序设计语言。

其它还有 PROLOG 语言和 LISP 语言（用于人工智能研究），C 语言（用于系统程序设计）等。

实际上，无论哪一种高级语言，计算机都不能直接执行，只有把它翻译成机器指令才能执行它。这个工作比汇编语言程序代真复杂得多。汇编语言的一个指令对应于一条机器指令，而高级语言中的一条指令（通常称语句和命令），往往对应若干条机器指令，将高级语言写的程序（源程序），转换成机器语言程序（目标程序），有两种不同的方式。

(1) 编译方式(compilation)。将源程序全部译成机器指令后，再将指令组成的目标程序交给计算机执行。

(2) 解释方式(interpretation)：它不是生产出目标程序再交付执行，而是边翻译边执行，翻译一句执行一句，不产生整个目标程序。

编译方式有如笔译，一次翻译出整个译文。解释方式如同口译，说一句译一句。用编译方式得到的目标程序执行速度快，但占计算机的内存空间大；解释方式执行速度慢，但占内存空间小。多数高级语言都采用编译方式。True BASIC 既可以用解释方式也可以用编译方式进行代真。

将高级语言程序转换成机器语言程序的翻译程序称为编译程序(compiler)或解释程序(interpreter)。

不是在任何一种计算机上都能随意使用任何一种高级语言。只有配备某一种语言的编译（或解释）程序，才能运行这种高级语言程序。例如，想在某台计算机上运行一个 True BA-

SIC 程序，必须保证计算机系统中配有 True BASIC 编译程序。在运行某一高级语言程序之前，先把它的编译程序（或解释程序）输入计算机内，由它对源程序进行编译。

高级语言的出现，使人们不必深入了解计算机内部结构和工作原理，也不必背诵枯燥无味的指令，而集中全力去研究算法。尽管运行一个高级语言程序比运行一个机器语言程序要慢（因为它要先经过翻译），而且由机器统一翻成的机器指令有时不如人工直接编写某一确定问题的机器指令程序紧凑，占用计算机内存空间大。但是，由于计算机技术的发展，计算机的运行速度和内存量迅速提高，其缺点与其在提高程序的编写效率和质量方面的巨大贡献相比，已是微不足道。

4. 非过程化的高级语言：上面介绍的高级语言，属于"过程化的高级语言"，用它编程序必须指出每一步应如何进行的全过程。即用高级语言去实现一个算法，既要解决"做什么"，又要解决"如何做"的问题，而且"做什么"是通过"如何做"体现出来的。程序设计时，设计者要详尽地规定计算机操作的细节（例如，计算某一值，打印某一变量值，进行某一条件判断等），将"做什么"逐步细化到"如何做"中。这使一些不熟悉计算机的人还感到不便，希望只指出"做什么"就行了，以使人的劳动减少到更低的限度。

70 年代后期，创造了"非过程化高级语言"（如 SQL、dBASE、FORTH、QBE 等），即第四代程序设计语言。使用这类语言编程序不必描述为解决某问题所经历的过程，只需指出"做什么"即可，至于"如何做"的细节由计算机系统自己去处理。例如，只要指出"把全校学生中小于 18 岁的女生中的团员名单打印出来"，计算机就能执行，并打印出所需结果。显然，这又大大前进了一步，为更多的人使用计算机提供了方便。

目前，国内外在计算机上运行的大多数仍然是过程化语言的程序，即用第三代计算机语言编写的程序。同时，学习用过程化语言编写程序，是目前学习计算机科学知识的一项基本训练。至于有关非过程化语言的知识，请阅读其它专门书籍。

1.4.3 程序设计语言的功能

各种程序设计语言千差万别、风格各异，但一般来说都应能提供以下四种功能：

（1）数据描述，即它对数据的使用有何规定以及如何描述它们。如 True BASIC 把数据分为数值与字符两类，这两类数据的存储与表示方式不同。

（2）操作运算，即它能直接和间接地提供哪些种类的运算。如 True BASIC 提供了算术、逻辑、字符串运算以及一些函数。

（3）程序控制，即它如何实现程序的顺序、选取、循环以及其它控制。如 True BASIC 用 IF——THEN——ELSE 实现选取控制，用 DO——LOOP 或 FOR——NEXT 实现重复控制。

（4）数据传输，即它允许使用哪种方式来描述数据的传输。如 True BASIC 中用 LET 表示把运算结果从运算器送入存储器，用 READ 表示把一个内存数据区中的一个数据读到一个变量所代表的内存单元中，用 INPUT 表示从键盘输入一个数据到内存，用 PRINT 表示从运算器把数据传送到显示器或打印机等。

学习一种程序设计语言应当着重了解它是如何提供这四种功能的。具体可以从以下三方面入手：

（1）语言的基础是一组记号和规则。根据规则由记号构成的记号串的整体就是语言。学

习一种语言时，应注意语言规定的各个记号之间的组合规律，即它的语法(syntax)。例如写 print 是正确的，而写 prnit 是不符合语法规则的。

(2) 了解按照各种方法所表示的各个记号的特定含义，即语言的语义(semantics)。例如，print 代表“打印”，而不代表“赋值”，这就是语义的规定。

(3) 了解用该语言所描述的程序与使用该程序的人之间的关系，即该语言的语用(pragmatics)。也就是说，人们如何正确利用语言达到特定的目的。

1.4.4 程序设计语言的使用

使用程序设计语言解题，一般要经过以下几个步骤：

1. 算法设计和编码：遇到一个要用计算机求解的问题时，首先应分析题目及用户要求，然后进行算法设计。算法设计是一个逐步细化的过程。即把算法描述成为不同的抽象级，一开始首先立足于全局性的“做什么”上去抽象，然后逐步把它具体化。

为了能清晰地表达算法的逻辑结构，便于交流，一般不应立即使用程序设计语言来描述算法。而且使用程序设计语言描述算法，必然涉及一些问题和语言的细节，干扰了逐步细化过程中在各个抽象级上独立地研究问题。通常是先给出对问题本身比较自然的、很可能是用数学或自然语言(或类似自然语言，如结构化的汉语或英语)描述的抽象算法。然后可以用专门的算法描述工具(如 N-S 图)将其细化，进入下一个抽象级，不断细化，直到足够详细时，才把它用程序设计语言代替。把已设计好的算法翻译成用某一特定的程序设计语言所编写的源程序的过程称为编码(coding)。

2. 录入和编辑：录入就是把编码得到的源程序通过终端打入到计算机内保存起来。在录入过程中或后来进行的过程中发现错误可以用编辑程序进行修改，如删去一行或一个字符、插入一行或一个字符等。

3. 编译：在编辑方式下录入的源程序被保存在外存中。它是不能直接执行的，在交给计算机执行前，应先把它编译(compilation)成为目标程序。如图 1.45 所示。在编译过程中，编译系统将检查程序中有无语法错误，如有，则输出“错误信息”。这种称“编译错误”(compilation error)。

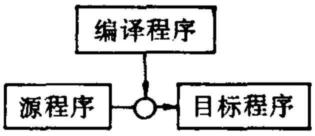


图 1.45

经编译得到的目标程序还可能要进行一些连接或装配工作(将不同的程序单位以及库程序连接起来，装配成一个可供执行的目标程序块)。连接和装配工作在有的系统中由编译程序自动进行，如 True BASIC；有的系统则需单独进行，由连接程序连接，如 FORTRAN 等。

4. 执行：经编译及连接得到的目标程序便可以执行了。一个程序的运行除计算机外还需三方面的支持：

(1) 操作系统，它是能控制和管理计算机硬件和软件资源(如存储器分配，输入/输出管理等)、合理地组织计算机的工作流程、为用户提供方便的程序的集合。常被称为计算机的“管家”。

(2) 语言系统。

(3) 程序所需要的数据。

程序执行的最后结果是给出问题的解答。程序中如果还有错误(包括语法错误或逻辑错

误)或输入数据有错误,在运行过程中,会输出一些错误信息,这就是“运行错误”(runtime error)。程序人员应该根据编译和运行的情况不断地修改程序,直到最后得到正确结果为止,这个过程称为“调试程序”(debugging)。

解释型语言是边执行边解释的,因此,上面步骤中的第 3 步(编译)和第 4 步(执行)是同时进行的。

True BASIC 源程序既可以以解释方式执行,也可以以编译方式执行。

习　　题

1.1.1　若你需要乘火车去外地办一件急事,请你把从自己房间出发到坐在火车内的一段算法用汉语写出来。

1.1.2　你会做菜吗?请你写一个算法,把你做的拿手菜的过程用汉语描述出来。

1.1.3　用汉语写一个你如何过星期天的算法。

1.1.4　“清零”是指把存储器单元的内容置为“零”。计算机开机后,通常先要对存储器清零。

某天,程序员甲开机后做了如下操作:

200⇒2001
10⇒2002
2⇒2003
(2001)*(2002)⇒2004
(2003)+(2004)⇒2005
(2005)+(2002)⇒2001
(2001)−(2002)⇒2002
(2002)−(2003)⇒2006
(2008)+(2003)⇒2007
5⇒2010

试问,经过上述操作后,存储器内 2000~2010 各存储单元的内容是什么?

1.1.5　辅助存储器有什么作用?

1.1.6　有人说:“人工解题的最好方法不一定是计算机解题的最好方法”。对吗?为什么?

1.1.7　为什么说:“计算机科学就是研究算法的科学”?

1.2.1　将你设计的 1.1.1、1.1.2、1.1.3 题中的算法分别用流程图和 N-S 图表示。

1.2.2　为什么要规定算法的三种基本结构?

1.2.3　下面是打印出最初 N 个素数的算法,请用 N-S 图表示。

(1) P 置初值 2,K 置初值 1,P_K 的初值为 P,S=0

(2) 打印 P

(3) 使 P+1⇒P

(4) 当 K<N 时,重复做下面的工作,否则转(5)

(i) 2⇒i

(ii) 当 $i \leqslant \sqrt{P}$ 时,重复做以下工作:

(a) $\frac{P}{i}$的余数⇒r

(b) 如 r≠0,使 i+1⇒i

否则 S=−1

(iii) 若 S=0,则打印 P

(iv) K+1⇒K,P⇒P_K,S=0

(v) P+2⇒P

(5) 结束

请弄清楚这个算法。P 是待判断的数。K 表示已找到的素数的个数,P_K 是第 K 个素数的值。S 作用“标志”,当 S=−1 时表示 P 能被某一数整除,P 不是素数。

1.3 用逐步细化的方法设计下列各题的算法(最后用 N-S 图描述)。

1.3.1 计算 $S=1+2+2^2+2^3+\cdots+2^{15}$。

1.3.2 计算 $S=a_0+a_1x+a_2x^2+\cdots+a_{10}x^{10}$。

1.3.3

(1) 有 10 个火柴盒排成一列,每个火柴盒中装入一张写着一个数的纸片,请将这 10 张纸片按所写数的大小顺序排列。排列过程必须按下列规则进行:

- 不能移动火柴盒;
- 可以借助其它火柴盒;纸片只能在火柴盒之间移动,且每个火柴盒中最多只许放一张纸片。

你能设计几种排列方法?哪一种排序方法使用的辅助火柴盒最少?哪一种排序方法移动纸片数最少?

(2) 排好序后,把 10 个火柴盒全部套好,使人不能看到火柴盒内的数,然后给定一个数,要你把装这个数的盒找出来。你能设计几种找法?哪一种方法打开火柴盒的次数最少?

1.3.4 求方程 $ax^2+bx+c=0$ 的根。

1.3.5 百鸡问题:“今有鸡翁一值钱五,鸡母一值钱三,鸡雏三值钱一,百钱买百鸡,问鸡翁、母、雏各几何?”

1.3.6 用一元人民币兑换一分、二分、五分的硬币。你可以设计几种不同的兑换方法?

1.3.7 给定一个年份,判断是不是闰年?

1.3.8 李白买酒:李白无事街上走,提壶去买酒。遇店加一倍,见花喝一斗,五遇花和店,喝光壶中酒。试问李白壶中原有多少酒?

1.3.9 某人用一个 7 斤和一个 11 斤的两个盛水器到池边取水,如何量才能只取回 2 斤水?

1.3.10 验证:任何一个大偶数都是两个素数之和。请将 6~100 之间的偶数表示为两个素数之和(如 6=3+3,8=3+5…)。这就是著名的哥德巴赫(Gold bah)猜想命题。

1.3.11 水分手椰子:有五个水手带一只猴子来到南太平洋的一个岛上,发现那里有一大堆椰子。由于旅途的颠簸,大家都很疲倦,很快就入睡了。第一个水手醒来后,把椰子分成五堆,将多余的一个分给猴子,他藏起一堆便又去睡了。第二、第三、第四、第五个水手后来也曾陆续起来,把见到的椰子和第一个水手一样重新分为五堆,把多余的一个给猴子,自己藏起一堆,又去入睡。天亮之后,大家一齐起来,把余下的椰子重新分为五堆,每人一堆,恰好余

一个又给了猴子。试问原先有多少椰子？

1.3.12 若口袋中放着12个球，3红3白6黑。要从袋中取出8个球，问有几种取法。

1.4.1 名词解释

(1) 位、字、字节、存储单元、存储地址；

(2) 机器语言、汇编语言、过程化语言、非过程化语言；

(3) 编译程序、解释程序、汇编程序；

(4) 语法、语义、语用。

1.4.2 一般说来，一种程序设计语言能提供哪些功能？

第二章 True BASIC 程序设计初步

2.1 概 述

2.1.1 True BASIC 简介

BASIC 是英文 Beginner's All—purpose Symbolic Instruction Code(初学者通用的符号指令代码)的缩写。它是由美国 Dartmouth 学院数学系的 John. G. Kemeny 和 Thomas E. Kurtz两位教授遵照学院把计算机变成所有学生都容易掌握的工具"的旨意,于 1963～1964 年间为初学者开发的一种小型程序设计语言。

BASIC 语言是基于下面的宗旨设计的:它应该使初学者容易学习;它应该是一种通用语言,可以编写各种程序,实现各种功能;它能提供人—机会话功能,能给用户提供清晰的出错信息;它对小程序能很快响应;它不要求用户了解机器的硬件结构,并把用户与操作系统隔开。这种"与用户友好"的设计思想给用户创造了一种良好的学习与工作环境。这正是 BASIC 语言的强大生命力之所在,使它能很快冲出校园,成为一种实用的程序设计语言,被广泛应用于社会的各个领域,并在应用中被扩展、强化。最初的"基本 BASIC"只有 17 条语句(1964 年 5 月 1 日问世时只有 14 条语句)。到现在,它不仅仅是初学者的良师益友,而且已经成为可以广泛应用于数值计算、数据处理、实时控制、绘图、游戏等多种领域,应用最广泛、最受欢迎的一种通用程序设计语言。国内外许多应用软件是用 BASIC 语言编写的。前不久有人调查,在波士顿计算机协会中 80%以上的成员使用 BASIC 语言编写程序。

1971 年是 BASIC 语言发展过程中的一个重要时期。这一年,BASIC 语言已经成为一种相当稳定的通用语言,正当它开始向着多功能、标准化的方向发展的时候,计算机的硬件方面取得了划时代的成就——微处理器问世了。真是天赐良缘?BASIC 对硬件资源要求较低,又为广大用户所倾心,这些特点深得新兴的微计算机厂商的赏识;每种微计算机系统几乎毫不例外地首先配置了 BASIC 语言,有的还把它固化在内存中。微型计算机因体积小、功耗低、工作可靠、价格便宜等优点,得到了迅猛发展,并载着 BASIC 语言奔向人类社会的各个角落。微型计算机产品的激烈角逐,使得 BASIC 语言的功能不断强化,但也给它带来了各种问题。

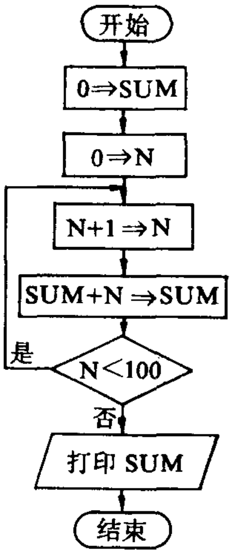
当初设计 BASIC 语言时,曾规定 BASIC 语言应不要求用户了解机器的硬件结构。可是在微计算机的角逐战中,各计算机厂商都急于卖弄自己机器的优势,给 BASIC 强加了许多依赖于硬件的特性。于是出现了众多的、与机器硬件相联系的 BASIC 版本,人们称之为 BASIC 的"方言性"。BASIC 的方言性破坏了它的通用性和可移植性,给标准化造成很大困难。

早期的 BASIC 语言多是解释执行方式。解释执行方式比较方便,而且又便于程序调试。但是,解释方式的执行速度较慢,这是使 BASIC 语言依赖于硬件的另一个重要原因。并且在实践中,常常出现这样令人恼火的情形:一个 BASIC 大程序运行了相当长的时间后,解释程序发现了程序中一个语法错误,用户不得不在纠正错误后,重新从头运行程序。

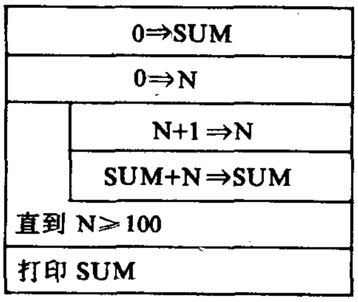
BASIC 语言的另一个弱点是它的非结构化。结构化程序设计是 60 年代后期，人们为对付日益严重的“软件危机”而提出的一种程序设计方法。这种方法提出后经过激烈论战，逐渐为计算机界承认。结构化程序设计要求有结构化的程序设计工具。于是人们一方面致力于新的结构化程序设计语言的研制，另一方面也致力于对现有程序设计语言的结构化改造。

使用行号是 BASIC 语言的一大特色。它要求每个程序行都用一个行号开始并作为该行的标识。

例 2.1.1 求 $\sum_{i=1}^{100} i$ 的 BASIC 程序。可先画出图 2.1 所示的流程图。用 BASIC 语言描述



(a)



(b)

图 2.1

所示算法，可以得到如下 BASIC 程序：

```
0 REM A basic program(注释)
10 REM Compute: SUM=1+2+…+100(注释)
20 SUM=0
30 N=0
40 N=N+1
50 SUM=SUM+N
60 IF N<100 GOTO 40
70 PRINT “Sum=”; SUM
99 END
Ok (提示符，说明机器处在命令状态)
RUN(执行命令)
Sum=5050(程序执行结果)
Ok
```

这里 0，10，20，30，40，50，60，70，99 都是行号。原则上，按行号的顺序（由小到大）执行各语句，执行到 60 行时，若满足“N<100”条件，则转去执行 40 行，40～60 行是一个直到型循环。

行号给程序的调试提供了很大方便。在行号的支持下，BASIC 允许程序员信手使用 GOTO 型语句把流程从一行转向到程序内的任何一行。这就容易导致程序呈现图 1.12 所示的 BS 型状态。BASIC 缺少有效地代替 GOTO 语句的控制结构，已有的控制结构也难以满足结构化程序设计的要求。

面对 BASIC 的这些缺点，人们已从多方面对它进行了改进，如制定 BASIC 标准，推出编译 BASIC 版本，改良并增加结构化的控制结构等。然而 BASIC 的两位创始人却不满足于这样一些零敲碎打的改良。1983 年，G. Kemeny 和 E. Kurtz 已经觉察到 BASIC 语言在发展中出现的问题的严重性，并认为已经到了用一个完美的设计、标准的版本来取代泛滥成灾的各种版本的 BASIC 的时候了。1985 年 11 月，Kemeny 和 Rurtz 推出了集 BASIC，FORTRAN 与 PASCAL 等语言的优点之大成的新的一代 BASIC 语言，并将它称为"真正的" BASIC——True BASIC。

True BASIC 严格符合美国国家标准(ANSI BASIC——1984 年在 Kurtz 主持下讨论并制定)是一种结构化、模块化的语言系统，它同时提供解释方式与编译方式，在保持 BASIC 原来的优点的基础上，进一步提高了程序的清晰度，有了很好的可移植性并在功能上作了较大的扩充，能够提供比普通 BASIC 大得多的用户空间，并且有了更强的编辑功能。这两位 BASIC 的创始人坚信，True BASIC 将开辟 BASIC 的新纪元。

2.1.2 True BASIC 程序的组成

True BASIC 程序是用 True BASIC 语言描述的算法。在详细说明如何用 True BASIC 语言编写算法之前，先了解一下 True BASIC 程序的格式。

在例 2.1.1 中，给出了求 $\sum_{i=1}^{100} i$ 的 BASIC 程序。这一程序在 True BASIC 状态下也可以运行，但是 True BASIC 更提倡一种不用行号的程序格式。

例 2.1.2 求 a+|b|的 True BASIC 程序。

```
! A True BASIC program for computing of a+|b| } (注释)
!                                             }
INPUT PROMPT"A,B=": A,B
IF B>=0 THEN
   LET C=A+B ! B>=0
ELSE
   LET C=A-B ! B<0
END IF
PRINT "A+|B|="; C
END
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

* * * 线以上为编辑窗口，是屏幕上书写及修改程序的区间；以下是背景窗口，是键入命令、显示结果的区间。在屏幕上，编辑窗口与背景窗口之间用一行"*"号分隔，也有的版本以一条直线分隔。本书后面均采用直线。

Ok. run(在命令状态下键入执行命令)

A,B=5,9(输入 a,b 值)

A+|B|=14(程序执行结果)

Ok,run(执行结束,回到命令状态,又键入执行命令)

A,B=5,-9

A+|B|=14

Ok.(执行结束,又回到命令状态)

例 2.1.3 求两个正整数 m 和 n(m>0,n>0,m>n)的最大公约数的 True BASIC 程序。

由图 1.28(a)所示算法可以得到如下 True BASIC 程序:

```
! Program of Euclid.
!
INPUT PROMPT"m,n=": m,n
LET r=mod(m,n)
DO WHILE r<>0
    let m=n
    let n=r
    let r=mod(m,n)
LOOP
PRINT "gcd="; n
END
```

```
Ok. run
m,n=35,21
gcd=7
```

根据图 1.27(b)所示算法可以得到如下 Ture BASIC 程序:

```
! Program of Euclid.
!
INPUT PROMPT"m,n=": M,N
DO
    LET R=Mod(M,N)
    LET M=N
    LET N=R
LOOP UNTIL R=0
PRINT"gcd="; m
END
```

```
Ok. run
m,n=35,21
gcd=7
Ok. run
m,n=56,72
gcd=8
```

Ok.

通过上述程序，对 True BASIC 程序有了一个直观认识。一般说来，True BASIC 程序应按如下规则组成：

1. 程序行：一个 True BASIC 程序由若干程序行组成。这里所说的"行"是指"逻辑行"，而不是指显示屏上以及写或用打印机打印在纸上的一个"物理行"。True BASIC 规定在一行内最多可以有 32767 个字符(包括字母、数字、空格或其它字符)。

每个程序行可以由行号、语句和回车键三部分组成，格式如下：

〈行号〉〈语句〉〈回车键〉

True BASIC 虽然允许使用行号，但更希望程序员不用行号。不使用行号，就迫使程序员不能使用以行号为依托的 GO TO 型控制，严格按结构化原则构造程序。在有行号的情况下，程序员依然可以使用旧 BASIC 提供的 GO TO 型控制结构，这主要是考虑到与旧 BASIC 的兼容，使那些用旧 BASIC 写的程序也能在 True BASIC 支持下运行。为了使读者养成良好的程序设计风格，本书不准备介绍带行号的 True BASIC 程序的设计方法。这一点旧 BASIC 的老用户也不会反对，因为使用行号的方法他们早已精通了。所以我们认为每个 True BASIC 程序行都由如下两部分组成：

〈语句〉〈回车〉

键入的每个程序行都以一个回车操作而告结束。在本书的程序中一般不写出回车操作。

2. 语句：语句是程序行中最核心的内容。True BASIC 程序中的每个语句都是 True BASIC 能接受的一条指令。每个语句都要用一个语句关键字开始。关键字用来区分不同的语句，告诉 True BASIC 执行不同的功能，如前面几个程序中使用的 LET，IF，THEN，ELSE，DOWHILE，LOOP，DO，LOOP UNTIL，PRINT，END 等都是语句关键字。True BASIC 不限制语句关键字的大小写。前面把语句关键字写成大写，是为了使程序清晰一些。后面将不再做这种区分，以提高输入的速度。要想得到语句关键字为大写的程序清单，只须调用一下"FORMAT"功能，便可将内存中的程序中的语句关键字全部改为大写了。

True BASIC 程序通常由基本操作语句(如 LET，PRINT，END)、程序控制结构语句(如 IF——THEN——ELSE——END IF，DO WHILE——LOOP，DO——LOOP UNTIL 等)以及说明语句组成。

True BASIC 规定，一行只能写一个语句，一个基本操作语句或一个说明语句只能写在一行内，控制结构可由一个语句或一个语句组给出，可以占用多行，如例 2.1.2 中的选择结构占了 5 行。

3. END 语句：任何 True BASIC 程序都应包含一条 END 语句。当执行到 END 语句时，程序结束。

4. 注释：上述几个程序中，语句关键字"REM"(也可用"！"代替)后面的内容，称为注释。注释是程序的不可执行部分，它对程序的运行结果和速度不会产生任何影响。它的主要作用就在于帮助人们阅读并理解程序，提高程序的清晰性，以便正确地使用和维护程序。注释可分为序言性注释和功能性注释两种。

序言性注释通常置于每个程序的开头部分，其内容包括：

· 有关本程序(或程序块)的功能和目标；

· 本程序块与其它程序块的接口；

· 有关数据的说明，如变量名的定义、用途、约束及限制等；

· 开发人员名单、设计日期、修改日期等有关开发历史的说明。

例 2.1.4 序言性注释示例。

```
! ——————————————————————————————————————————————————
! * The program calculates the greastest common divisor      *
! * of two integer numbers using euclidean algorithm.        *
! * The program works until either of the input number is 0. *
!
!
! Programer: XX_XXX_XX; Programer: NO.: 930022
! Date: 1988.3      Task no: b01
!
! Variable: integer; i,j,k,gcd
!
! ……
```

功能性注释通常嵌在源程序中，用以作功能描述，对程序的局部功能作说明性注释。应该用一些空行或空格把注释与程序分开，使源程序更清晰。

例 2.1.5 功能性注释示例。

```
! These are"remark"statements.
! They must be all alone on a line.
!
! The"!"comments work about the same.
! However, they may also appear at
! the end of any statement.
!
! let a=1   ! like this…
end
```

源程序文卷中的注释，提供了程序开发人员与源程序的读者之间的一种沟通手段。过去人们对这一手段不够重视，也利用得不够。但是随着程序的规模不断增大，人们才逐步体会到使用它的好处，并普遍认识到它已是源程序的一个不可忽视的重要组成部分了。有人觉得写注释太麻烦。实际上注释可以在写程序的同时一起写出。只要养成了习惯，认识到写注释的重要性，就会觉得写注释将是一件必须做的工作了。目前，True BASIC 已经汉化，许多注释也可以用汉字写出。还有人觉得写注释会影响程序的执行速度。这一点也请读者放心，True BASIC 中附加的字符是不会影响执行速度的。所以完全可以放心地写易读性好的程序。当然，注释应当正确，因为不正确的或容易使人误解的注释比没有注释更坏。同时，注释也不是越多越好，因为太多的注释会占用较多的存储空间，也破坏了程序的简明性并增加了人的工作量和出错的机率。

2.1.3 True BASIC 字符集

True BASIC 字符是书写 True BASIC 源程序的基本元素。True BASIC 字符集包括：

- 26 个大写英文字母，26 个英文小写字母。
- 10 个数字字符。
- 特殊字符：如＋、－、＊、/、(、)、＝、，、·、$、%、!、&、空格以及机器能识别的其它字符。

在计算机内部，字符是用二进制码表示的。用二进制码表示字符的方法很多，在小型和微型计算机中流行的是美国标准信息交换码——ASCII 码(American Standard Code for Information Interchage)。附录 D 为在 IBM PC 上使用的字符集。其中第 1～第 128 个字符是标准的 ASCII 码字符，第 129～第 256 个字符是 IBM PC 上特有的字符，在其它机器上可能会不同。

在汉字操作系统支持下，True BASIC 还允许使用汉字。

2.1.4 使用 True BASIC

下面介绍在 IBM PC(长城 0520)上使用 True BASIC 的基本方法：

1. True BASIC 的启动与退出：True BASIC 要在 MS-DOS 2.0 或更高的版本的支持下工作。当机器运行在 MS-DOS 命令状态时，键入"hello"命令，便会把保存在当前盘中的 True BASIC 调入内存。

"Ok"是 True BASIC 的提示符。它告诉用户系统目前处于命令状态，可以输入程序或发布命令。

在命令状况下键入命令"bye"，便会退出 True BASIC，返回到 DOS 状态。

2. 窗口：窗口是在屏幕上开设的一块显示区域。在 True BASIC 命令状态下，屏幕被分隔成两个窗口，如图 2.2 所示。上面的窗口叫做编辑窗口(editing window)，可供用户书写、显示或编辑现行版本的 True BASIC 程序。下面的窗口叫做背景窗口(history window)，供用户键入命令，并在此显示运行结果。

3. 编辑源程序：编辑功能用来输入源程序以及对源程序中的错误进行修改。True BASIC 的编辑功能是在全屏幕状态下进行的。即在需要插入或修改某些字符时先用键盘操作把光标移到要插入或修改处，然后在光标所在处进行所需的编辑。表 2.1 是在 IBM PC 上进行全屏幕编辑时的光标移动键一览表，表 2.2 是常用编辑键一览表。

表 2.1

F1	进入编辑窗口	F2	返回背景窗口
←	左移一字符位置。	PgUp	向上翻页
→	右移一字符位置。	PgDn	向下翻页
↑	上移一行。	Tab	右移一字
↓	下移一行。	Shift＋Tab	左移一字。
Home	移至窗口顶部。	Ctrl＋←	移至行首。
End	移至窗口底部。	Ctrl＋→	移至行尾。

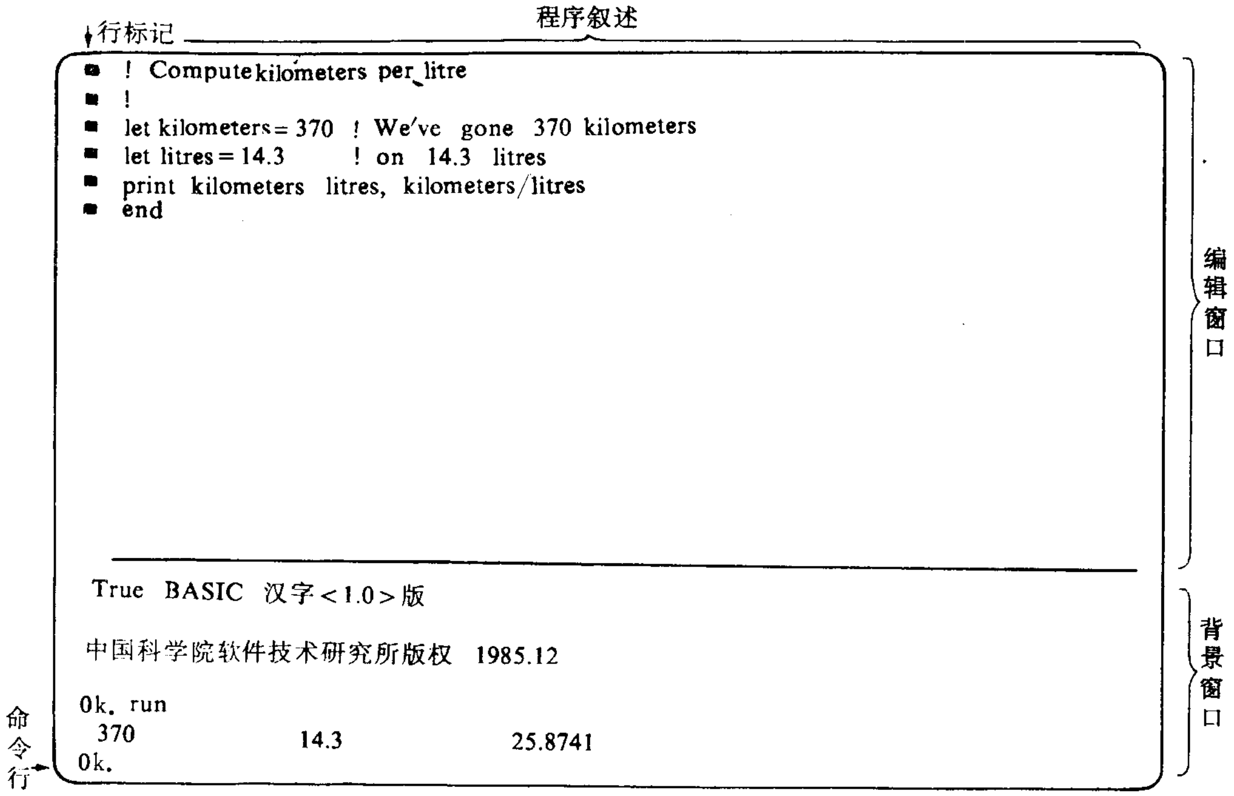


图 2.2

表 2.2

退格键	删除左边一字符。	F4	使行标志闪动，为删除、复制、移动、压缩或编辑命令作准备，也可解除当前行的标记。
Del	删除当前字符。		
Ctrl + Home	删除左边一个字(词)。	F5	在编辑窗口可以将有标记行模块复制到光标所在行的紧后面。
Ctrl + PgUp	删除当前一个字(词)。		在背景窗口，光标在命令行时可以复制上一行命令。
ESC	删除当前行中光标左边的全部字符。	F6	移动带标记的行模块到光标所在位置。
Ctrl + End	删除当前行中光标右边全部字符。	> 或 <	使标记行模块右移或左移一格。
Del	光标在行标志处时，删除当前行。	退格键	当光标位于行标志处时，将当前行连接到上一行。
F7	重新存入被删除的当前行。	Ins	插入或替换字符。
↵	在光标所在处插入一空行。		
F10	Help 帮助。		

当要建立一段新的程序时，必须先将内存中的程序清除，清除内存中的程序的命令是“NEW”。

4. 解释、编译、执行：True BASIC 允计解释与编译并存。解释状态跟踪能力强，用法简便、灵活，给程序的编制、调试和修改带来很大方便。一个熟练的程序员往往可以直接在机器上用边输入、边修改的方式开发一些小型程序，并能立即获得结果。编译方式是把源程序首先转换代真为目标代码。通常编译代码比源程序代码要小得多，程序运行的速度也能大大提高，而且所有的非法结构或调用的不合法错误在编译阶段就发现了。在发现这些错误的同时，还能提供清楚、易于理解的出错信息，这些信息不依赖于硬件来维持。

由于 True BASIC 允许解释与编译并用，用户可以先在解释状态下进行程序的输入与调试，然后将调试好的程序用一条“compile”命令再将它编译为可执行的目标代码。

无论是在解释状态，还是在编译状态，程序都是用同样的命令“run”来执行。程序运行的结果，True BASIC 会显示“Ok”以等待下一个命令输入。

5. 程序文件的保存与调用：True BASIC 允许用户用一个 SAVE 命令把内存中的 True BASIC 源程序或经过编译的目标程序以“文件”(又称文卷)形式保存在指定的磁盘驱动器中的磁盘内，以便以后需要时调用。SAVE 命令的格式为

```
SAVE [〈驱动器号〉:]〈文件名〉[.〈扩展名〉]
```

True BASIC 要在 DOS 2.0 的支持下才能运行，并且能处理所有 DOS 2.0 操作系统提供的文件名格式。DOS 2.0 规定文件名最长为 8 个字符，后面可以加长度不超过三个字符的扩展名。扩展名与文件名之间以圆点“·”相隔。可用作文件名或扩展名的字符有：

- 英文字母。
- 数字 0～9。
- 特殊字符：$,# ,&,@,!,%,(,),－,{,},‘,—,’。

一些合法的文件名：TAXES，INVOICE#，TAXES。TRU，1984.JAN，DUDGET.%。

如果省略扩展名，True BASIC 将会按各文件的类型给它自动加上一个合适的扩展名。如给 True BASIC 源程序文件加上“.TRU”，给编译文件加上“.TRC”。

存在磁盘驱动器内磁盘上的 True BASIC 文件可以用 OLD 命令装载到内存中。OLD 命令的格式为

```
OLD [〈驱动器号〉:]〈文件名〉[.〈扩展名〉]
```

省略驱动器号时，隐含指当前驱动器。

在一个磁盘中不能存储两个同名文件。当将一个文件修改后，要重新存入时，应使用 REPLACE 命令。格式如下：

```
REPLACE〈文件名〉
```

这样 True BASIC 才能将内存中的程序写到同名文件的位置上，并取代原来的文件。

在本节中只简单说明将源程序保存在磁盘上以及从磁盘上将源程序输入到内存的方法，以便读者上机操作时有所遵循。关于“文件”的概念和使用不拟在本节中全面介绍。

2.2 数据描述

2.2.1 数据类型

在计算机科学中，数据(data)是指能由程序处理的对象。早期计算机主要用于数值计算，当时的数据与数值数据(numerical data)属同义词。50年代中期，计算机开始用于数据处理。在数据处理中，不仅要处理数值数据，而且要处理非数值型的数据(例如文字数据)，于是人们又引入了非数值的数据——字符型数值，使数据概念的外延得到扩充。

在高级语言程序中，每一个数据都属于一个特定的类型。在同一个高级语言中，不同类型的数据的表示、存储及操作的种类是不相同的。类型信息可用于检查和防止程序中无意义的结构，也可用于确定怎样表示及用哪一类操作对该数据进行处理。每一组高级语言都有一组它所允许的数据类型(data types)。一般说来这些数据类型可包括如下三类：

1. 基本类型：True BASIC 可以提供的基本类型有两类：数值型和字符型。

数值型数据一律用十进制表示，最大有效位为10位(在某些机器上可更长，如在带8087的机器上为16位)，取值范围为1×10^{-99}(也可表示为1e−99)到1×10^{99}(或表示为1e+99)。当要计算的数大于1×10^{99}时，机器将给出出错信息"overflow"；当计算出的数小于1×10^{99}时，True BASIC 就用0来代替它。数值数据在机器内都按二进制数形式存放，每个数值型数据在内存占8个字节。

在程序中每个字符数据(character data)是一串用双引号作起止界的字符序列，称为字符串(character string)。如：

"This if a string literal!"

"Personal computer"

"* * * * * MENU * * * * *"

等，都是字符串。字符串中包括的字符数称为字符串长度。True BASIC 中的字符串最少占4个字节，最长可达32K字节(有的机器还要长)。没有字符的字符串，称为空串，用" "表示。True BASIC 规定一个字符串中也可以包含双引号，在字符串中，用连续两个双引号(" ")代表一个双引号(其它 BASIC 不允许)。如："x""y"表示"x"后面跟一个双引号，再后为y。" "" "表示只包括一对双引号的字符串。

字符串数据在机器内部按 ASCII 码方式存放。IBM PC 上使用的字符集及对应的 ASCII 码见附录 D。

2. 构造类型：构造类型是由已知类型如基本类型按一定的规则构造起的有结构的数据类型，例如数组(array)、记录(record)、文件(file)和集合等。True BASIC 可以提供数组、记录和文件等三种构造类型的数据结构。

3. 指针类型：指针类型主要用于构造各种形式的动态数据结构，如链表、堆栈、队列、树、有向图等。这种数据类型很有用处。True BASIC 不直接提供这种数据类，但是可以用数组来模拟它。

在进行程序设计时，不仅要描述数据本身的属性，往往还要反映数据之间的联系。反映

数据的属性及其数据间联系的数据模型，称为数据结构。选择合适的数据结构，可以使解题算法简化并且更有效。所以数据结构的建立是程序设计的一个重要内容。基于这一思想，美国计算机科学家 Niklaus Wirth 提出了一个著名的公式：

算法＋数据结构＝程序

来表达程序的实质。他说："程序就是在数据的某些特定的表示方式和结构的基础上，对抽象算法的具体描述。"

在本书以后各章中将逐步介绍如何设计合适的数据结构以及如何用 True BASIC 实现它。

2.2.2 常量与变量

1. 程序中常量与变量的实质：常量(constant)与变量(variable)是数据的两种形态。常量是在程序运行期间始终保持不变的数据，而变量是在程序运行期间其值可以改变的数据。计算机程序中变量这个概念与普通数学中使用的变量的含义是不同的。在普通数学中，变量隐含着"未知"。而在计算机程序中，在程序执行的每个瞬间变量的值都是确定的、已知的，但在程序执行的整个过程中它的值是可以变化的。

我们知道，计算机中的信息是存储在存储器中的，当定义某个变量时，就是用这个变量名代表某一个存储单元，同时指定了该存储单元按变量所具有的类型去存储数据。至于什么时刻，所存的具体数据是什么，由程序操作决定，即程序执行过程中给各变量提供具体数据。图 2.3 为对变量 A 的操作过程示意：一开始 A 中存的数为 0；接着程序中某一语句把数值 3 送给它，它的值便变为 3 了。再把 5 送给它，它的值又变成 5 了。这一操作过程用 True BASIC 描述如下：

```
LET A=0
LET A=3
LET A=5
PRINT "A="; A
END
```

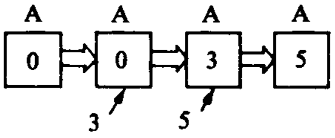


图 2.3

变量在某时刻的值，称为变量的现行值。程序中各语句对变量的引用，实际上是对其现行值的引用。程序功能的实现是对程序中变量进行加工的结果。

BASIC 能自动给所有变量以初值 0，在使用变量时，如果还没有给变量送过一个值，则它就是零。

常量与变量不同，在于存储形态不同。变量是独立存储的，而常量是作为某语句的一部分被存储的。一旦该语句被执行，就不允许再对它进行存取操作了。也就是说如果在程序中三处出现常数 3.5，它要占三个存储单元，三个 3.5 并不在同一个存储单元中。因此程序中要重复使用某些数值时，最好将它以变量的形态存放，以便节省存储单元。

2. 变量名：由上述讨论可知，变量代表在程序执行过程中其值可以改变的存储单元。这个存储单元的名字称为变量名。它是变量的标识，变量命名规则依 BASIC 版本而异。True BASIC 中变量的命名规则如下：

- 变量名是以字母打头的字母、数字或下划线组成的字符串；
- 变量名的长度最长可达 31 个字符；

· 在变量名中不区分大写与小写字母；

· 不允许使用保留字作变量名。True BASIC 的全部保留字如表 2.3 所示；

表 2.3

关键字	函数名	
ELSE	DATE	MAXNUM
ELSEIF	DATE $	PI
IF	DET	RND
NOT	EXLINE $	TIME
PRINT	EXTEXT $	TIME $
REM	EXTYPE	

· 变量的类型以变量名的最后一个字符来区分。当变量名的最后一个字符为“ $ ”时，该变量为字符串变量，否则为数值变量。字符串变量同数字变量一样，其值(具体的字符串)在程序执行过程中也是可以更动的。

表 2.4 为 True BASIC 变量用法示例。为了提高程序的清晰度，变量名应按“见名知义”的原则命名，如年龄用“age”，名字用“name”等。这样在程序中使用大量变量时，就不致让读程序的人在记变量名上浪费时间，也不致弄错。

表 2.4

合法形式	非法形式
amount	am ␣ ount
name_of_student	name # of # student
a	3a
string $	date
worker_1385	1385worker

3. 常数的表示：True BASIC 只能接受符合语法规则的常数。

字符串常数必须用两个双引号括起来。字符串中可以包含空格。空格也是一个字符，如“␣ AB ␣ C”不同于“ABC”，也不同于“AB ␣ C”“␣”表示一个空格。在字符串中大小写字母是有区别的，例如“joy”和“Joy”是不同的两个字符串。

数值常量的数字中不能包含空格或逗号，例如 12345 不能写成 12,345 或 12 ␣ 345。数值常量有两种表示格式：定点格式与浮点格式。

定点格式也就是日常使用的十进制记数法。它由一个正负号(正号可省略)和数字两部分组成；数字部分可以带一个小数点，也可以没有。表 2.5 为 True BASIC 定点格式数值常量示例。

表 2.5

合法形式	非法形式	合法形式	非法形式
−6	−6−	−0.003	100 ␣ 000 ␣ 000
+1002	1,002	25.	3/2
321.33	1.2.3	.78	8^2

浮点格式也称常指数格式或科学记数法。它由正负号、数字部分、指数部分三部分组成。

指数部分由字母“e”或“E”开头，后面跟一个正号(表示指数为正，这时也可省略正号)或负号(表示指数为负)和一位或多位数字(表示指数)。指数表示法适合表示非常大或非常小的数，如 3000000000 可以写成 3e9 或 3e＋9 或 3E9，0.00000003 可以写为 3e－8 或 3E－8。

用定点格式表示一个常数，其小数点的位置是固定的，如 12.38，在个位数字之后有一个小数点。而用浮点格式表示一个数时，其小数点位置不是固定的，而是可以变化的，如 12.38，可以表示为 12.38e0，也可以表示为 1.238e1，0.1238e2，或 123.8e－1，1238e－2 等，都代表同一个常量。

2.3 True BASIC 表达式

True BASIC 允许进行算述运算、逻辑运算和字符串运算。这里先介绍算术运算。

2.3.1 算术操作符

True BASIC 提供了加、减、乘、除、乘方等五种算术操作。表 2.6 为这五种算术操作的操作符及用法示例。读者应当注意区分它们与普通算术中的符号及用法，如计算汽车每公升油的行程，不能写成

kilimetres÷litres 或 $\frac{\text{kilimetres}}{\text{litres}}$，而要写成 km/1

表 2.6

算术操作符	含　义	示　例
+	加(或正)	a+b
－	减(或负)	a－b
*	乘	length * width
/	除	km/litres
^	幂(乘方)	x^3

2.3.2 标准函数

函数是一种特定的运算。在程序中要使用一个函数时，只要写出函数名并给定一个或多个参数(数值或字符串)，它便能给出一个输出。在 True BASIC 内部直接提供用户使用的函数有四类：数学函数；三角函数；字符串函数；日期和时间函数。

这些函数在程序中的调用方式为一个函数名后跟随着括号内的一个或多个参数，这些参数之间以逗号相隔。表 2.7 为 True BASIC 提供的数学函数与三角函数一览表。字符串函数留在第五章介绍，日期和时间函数请参阅有关手册。

2.3.3 数学表达式

用符号 True BASIC 规定的算符、常量、变量、函数以及圆括号连接起来的式子，称为 True BASIC 表达式。书写表达式时，所有的字符都应排在同一水平线上。True BASIC 有三种类型的表达式：数值表达式、字符串表达式和逻辑表达式。

表 2.7

函　数	含　　义	应用举例	
		调　　用	返　　回
ABS(x)	$\|x\|$	Abs(89.3) Abs(0) Abs(−89.3)	89.3 0 89.3
DIVIDE (x,y,q,r)	x/y,商→q,余数→r	CALL Divide (17,8,q,r) CALL Divide (7.5,3.5,q,r) CALL Divide (4,20,q,r)	q=2,r=1 q=2,r=0.5 q=0,r=4
EPS(x)	返回一个"不影响 x 的值的小数",即告诉用户:机器表示 x 时的可能出现的误差。	在 IBM PC 机上 EPS(0)	 1.11254e−308
		EPS(0)表示所使用计算机上所能处理的最小正数。	
		EPS(1e+13)	0.03115(表示一个10 亿的数时,误差不超过 0.03115)
EXP(x)	e^x	EXP(0) EXP(1)	1 2.71828
INT(x)	返回不大于 x 的最大整数。	Int(−1.1) Int(1.6)	−2 1
LOG(x)	ln(x)　(x>0)	Log(1) Log(EXP(10))	0 10
LOG2(x)	$\log_2(x)$ (x>0)	Log2(4) Log2(0.5)	2 −1
LOG10(x)	$\log_{10}x$(x>0)	Log10(10) Log10(le25) Log10(.001)	1 25 −3
MAX(x,y)	返回 x,y 中的大数。	Max(11,13) Max(4,−7)	13 4
MAXNUM	返回所用机器上所能使用的最大正整。	对 IBM PC MAXNUM	 3.59539e+308
MIN(x,y)	返回 x,y 中的小数	Min(4,3) Min(10,−360)	3 −360

续表

函数	含义	应用举例	
		调用	返回
MOD(x,y)	返回 x 除以 y 的余数(y≠0)等价于 Mod(x,y)=x−y * Int(x/y)结果在 0 与 y 之间。	Mod(15,2) Mod(6.3,2) Mod(−1,2) Mod(14,2) Mod(7,−2)	1 0.3 1 0 −1
REMAINDER(x,y)	返回 x 除以 y 的余数(y≠0),x,y 可为小数余数的符号与 x 一致。	Remainder(8,3) Remainder(2.8,1) Remainder(−1,3) Remainder(8,−3)	2 0.8 −1 2
RND	返回一个伪随机数,0≤Rnd<1		
ROUND(x)	返回 x 经四舍五入的整数,等价于 Int(x+0.5)。	Round(4.28) Round(7.8666) Round(−8.3)	4 8 −8
ROUND(x,n)	返回 x 取 n 位小数后的近似值。若 n≥0,近似到小数点右边 n 位;否则近似到小数点左边 n 位。舍弃时按四舍五入处理(这一点与 TRUNCATE 函数不同)。	Round(492.839,2) Round(492.839,1) Round(492.839,0) Round(492.839,−1) Round(492.839,−2)	492.84 492.8 493 490 500
SGN(x)	返回 x 的符号:x>0,Sgn(x)返回 1 x=0,Sgn(x)返回 0 x<0,Sgn(x)返回 −1	Sgn(989) Sgn(0) Sgn(−4.8e−4)	1 0 −1
SQR(x)	返回 x 的平方根(x≥0)。	Sqr(2) Sqr(100)	1.41421 10
TRUNCATE(x,n)	用 n 对 x 的有效位数进行截取。 n≥0,取 x 的小数点右 n 位。 n<0,取 x 的小数点左 n 位前的数被舍弃的位上的数均赋以零,不四舍五入(这一点与 ROUND 函数不同)。	Truncate(137.291,1) Truncate(137.291,0) Truncate(137.291,−2)	137.2 137 100
ANGLE(x,y)	返回 x 轴与坐标(x,y)之间的夹角的弧度数(x 和 y 都为零时,出错)$-P_i \leq$ Angle(x,y)$\leq P_i$。	Angle(1,0) Angle(−1,0) Angle(0,1)	0 3.14159(P_i) 1.5708($P_i/2$)
ATN(x)	arctgx	Atn(0) Atn(P_i)	0 1.26263
COS(x)	cosx	Cos(0) Cos(P_i)	1 −1

续表

函　数	含　　义	应用举例	
		调　　用	返　　回
PI	返回 π 的值。	PI	3.1415926……
DEG(x)	把 x(弧度)转换成角度(x 总是以弧度为单位)。即使在程序中使用了 OPTION ANGLE DEGREES 语句,对 DEG(x)中的 x 的单位不起影响。	Deg(P_i) Deg(P_i/4)	180 45
RAD(x)	把 x(度)转换为弧度,即使程序中包含 OPTION ANGLE RADIANS 语句,对 RAD(x)中的 x 不起影响。	Rad(180) Rad(45)	3.14159(P_i) .785398(P_i/4)
SIN(x)	sinx	Sin(0) Sin(P_i/2)	0 1
TAN(x)	tgx(x≠P_i/2+n * P_i,n 为任意正整数)	Tan(0) Tan(P_i/4)	0 1

注:三角函数中的角度一般以弧度为单位。若程序中以 OPTION ANGLE DEGREES 语句说明,则在其后的三角函数的角度单位是度(0°～360°),而语句 OPTION ANGLE RADIANS 使其后所有的角度单位为弧度(0～2π)。DEG 与 RAD 函数不受这两条语句的影响。

表达式中执行算术运算的优先顺序是:函数→幂→乘或除→加或减。可以用圆括号改变运算的次序,内层括号中的运算比外层括号的运算优先级别高。

2.3.4 字符串表达式

字符串运算有两种:一种是连接,一种是取子字符串。

字符串连接的运算符为“&”。如字符串表达式:

“Sh” & “ang” & “hai”

产生的结果为字符串:

“Shanghai”

在字符串后跟一个[a : b](或(a : b)),便可以得到该字符串中的一部分——子字符串。其中 a 用来确定子串的起始位置,b 用来确定子字符串的结束位置。如字符串表达式:

“Wu” & “Shanghai” [2 : 4]

产生的结果为字符串:

“Wuhan”

若 a<1,则系统自动以 1 代替;若 b 大于整个串的长度,则用串的长度来代替(也可以用〈字符串〉[a : maxnum]来得到字符串中从第 a 个字符到串尾的子串);若 a>b,则产生一个空串。

在字符串表达式中,可以使用圆括号改变运算顺序。例如:(“Wu” & “Shanghai”)[2 : 4]代表先连接,后求子串,其结果为“uSh”。

关于字符串运算的进一步内容及逻辑表达式的概念,将在以后的章节中介绍。

2.4 数据传送

数据传送是程序设计语言提供的一个重要功能。True BASIC 提供有如下五种形式的数据传送方式。

1. 由运算器向显示器或打印机传送数据。这个数据是运算器对某一个或几个表达式运算的结果。这一传送操作用语句关键字 PRINT 表示，称为打印。PRINT 语句是 True BASIC 唯一的输出语句。

2. 由运算器向存储器传送数据。即向存储器中某一存储单元(变量)传送一个数据。这个数据是运算器对某一个表达式运算的结果。这一传送用语句关键字 LET 表示，称为"赋值"。

显然，PRINT 语句与 LET 语句除有数据传送功能外，还有运算功能。

3. 由存储器中已被程序定义的数据区向存储器中已被程序定义的变量中传送数据。True BASIC 用语句关键字 DATA 表示字义数据区，用 READ 实现传送，称为"读数据区"。

4. 由键盘向存储器传送数据。这一传送操作用语句关键字 INPUT 表示，称为键盘输入。

5. 由内存向外存，或由外存向内存传输数据。这一操作称为文件操作。因为数据在外存是以文件的形式存放的。

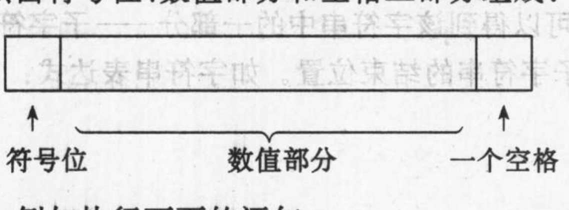
这一节主要介绍前面四种传送方式，即四种语句的使用方法。

2.4.1 PRINT 语句

PRINT 语句的作用是将运算器的运算结果传送到显示器或打印机。或者说，PRINT 语句的功能是输出表达式的值，即先对表达式的值运算，然后将所得的值输出。它的用法很灵活，这里只介绍它的基本用法。

1. 项输出格式：PRINT 语句把每个表达式作为一个输出项。表达式的类型不同，项输出格式也不相同。每个数值表达式输出一个数值项，每个字符串表达式输出一个字符串项。下面介绍这两种项的输出格式。

(1) 输出数值项：用 PRINT 语句输出一个数值表达式的值，可得到一个数值项。每个数值项由符号位、数值部分和空格三部分组成：



例如执行下面的语句：

print 135＋531

显示出的数值结果为

␣666␣(␣表示一个空格)

执行语句

print－3e15＊1.2e－5

后，显示出的数值结果为

－36000000000 ␣

需要说明以下几点：

① 一个不超过十二位的整数，True BASIC 中 PRINT 语句将把它做为一个整数按定点格式输出。如

48159

－325575654

② 如果输出的数是小数，而且小数点后的数字位数不超过六位，也按定点格式输出，输出时省略小数点最后的末尾零。如

983856

－7649.89

.00049

③ 除上述两条之外，将以如下格式按科学记数法输出：

[±]d.ddddde{±}dd

每一个“d”代表一个十进制数字，但小数点前的第一个“d”代表一个 1～9 的数字。小数点后的末尾零也不输出。如：－8.36e＋02，7.31653e－25。注意，不会输出以下的形式：－83.6e＋01，731.653e－27。

PRINT 语句只能输出最多六位有意义的数字。如需要大于 6 位的精确数字，可以使用 PRINT USING 语句。PRINT USING 的用法在第七章中再另作介绍。

(2) 输出字符串项：PRINT 语句能将字符串表达式的值按原样输出，后面不留空格，前面也不留空格。如

print “Wu” & “Shang hai” [2 : 4]

将输出下面的字符串项：

Wuhan

(3) 输出空行：PRINT 语句中没有表达式，即不包括任何输出项而是一个空的 PRINT 语句，将输出一空行。空行可以使输出的数据清晰。例如下面程序：

```
print “This example leaves a blank line”
print
print “Between two lines”
end
```

输出结果如下：

```
Ok. run
This example leaves a blank line
Between two lines
Ok.
```

2. 一个 PRINT 语句输出多个数据：一个 PRINT 语句允许输出多个表达式的值。这些表达式之间可以用“;”或“,”分隔，形成紧凑格式（也称自由格式）或分区格式（也称固定格式）。下面分述它们的用法。

(1) 紧凑格式：当一个 PRINT 语句中的表达式间以“;”分隔时，每一个“;”都使后面的

一个输出项紧靠前面的输出项输出。如语句

print “A+B=”; 5; “␣␣”; “A−B=”; −1

将能得到下面的输出：

A+B=␣5␣␣␣A−B=−1␣

(2) 分区格式：True BASIC 把每个输出行(物理行)都划分几个固定的打印区，每一区域的宽度是 16 个字符。当一个 PRINT 语句中的表达式间以“,”分隔时，每一个“,”都使后面的一个输出项在下一个打印区的开始位置输出。如语句

print “A+B=5”, “A−B=−1”, “A=2,B=2”

将输出下面的结果：

A+B=5	A−B=−1	A=2,B=3
←第一打印区→	←第二打印区→	←第三打印区→

若一个数据项太长，超出规定打印区宽度时，下一项将被下一个区输出，如语句

print “ABCDEFGHIJKLMNOPQ”, “OK!”

将输出下面的结果：

A B C D E F G H I J K L M N O P	Q	OK!
←第 1 打印区→	←第 2 打印区→	←第 3 打印区→

每个“,”跳一个区，n 个“,”跳 n 个区。当一个输出行被占满时，将从下一个物理行输出。如

print 1,2,3,4,5,6,7,8,9

输出结果如下(共分 2 行输出)：

␣1	␣2	␣3	␣4	␣5
␣6	␣7	␣8	␣9	
第 1 打印区	第 2 打印区	第 3 打印区	第 4 打印区	第 5 打印区

(3) 混合输出格式：一个 PRINT 语句中允许“;”与“,”混合使用。混合使用时，“;”与“,”所隐含的打印格式不变。如语句

print “A+B=”; 5,, “A−B=”; −1

将输出下面的结果：

A+B=␣5␣		A−B=−1
←第 1 打印区→	←第 2 打印区→	←第 3 打印区→

(4) 续行：当一个 PRINT 语句的末尾是一个“;”或“,”时，将使下一个 PRINT 语句的输出项按“;”或“,”隐含的格式，接在该 PRINT 语句的最后一项后面输出。如

print “A+B=”;

print 5,

```
print "A-B=";
print-1
```

将输出下面的结果：

A+B=␣5␣	A-B=-1␣

(5) 输出到打印机：通常 PRINT 语句是在屏幕上显示程序的输出。当要使用打印机打印程序的输出时，应在 PRINT 语句之前先打开打印机通道。

在程序中用 OPEN 语句打开打印机通道。这时 OPEN 语句的格式为

OPEN #〈通道号〉:PRINTER

其中通道号为 1～1000 之间的任意整数。如

```
open #2: printer
```

则指定 2 号通道为打印机通道。以后使用语句

```
print #2:〈表达式〉
```

便可将表达式的值输出到打印机，在纸上得到打印结果。

2.4.2 LET 语句

LET 语句称为赋值语句，它的功能是把运算器中的数据(即对一个表达式运算的结果)传送到存储器中一个存储单元。具体的说是"先运算后赋值"。它有两种句型格式：

LET〈变量〉=〈表达式〉

和

LET〈变量 1〉,〈变量 2〉,……=〈表达式〉

下面是一些 LET 语句的例子：

```
let e=2.718281828
let a,b=2
let x=e^a+b
let mouth $ ="Semtepber"
let abbrev $ =mouth $[1:3]&"·"
```

LET 语句的功能是首先计算出一个表达式的值，然后把这个值送给"="号左边的各变量。

要正确地使用 LET 语句，应注意以下几点：

1. 在 LET 语句中，"="称为"赋值号"，而不能称其为"等号"。它与等号形同而义不同。等号静态地表明其两边的表达式的值之间具有"相等"的关系。而赋值号是表明一种操作，将它右边的表达式的值送到其左边的各变量所代表的存储单元中去。因此，不能用等号的概念来理解赋值号。如

```
let t=t+1
```

表示把变量 t 的值加 1 后，再送到变量 t 中去。如 t 原为 2，则给上述赋值操作后 t 的新值变为 3。若把它当作等号，显然是无法理解的。

赋值号与等号的概念不同，用法也不同。等号两边都可以是表达式，两边对调后与原来

的表示等价。赋值号右边是表达式,左边必须是变量,两边不可以对调,即使两边都是变量,对调后所产生的结果也不同。如下面的两段程序的作用是不相同的。

程序段 1:

```
let A=1
let B=3
let A=B
print "A="; A, "B="; B
end
Ok run
A=3            B=3
```

程序段 2:

```
let A=1
let B=3
let B=A
print "A="A, "B="; B
end
Ok run
A=1            B=1
```

2. True BASIC 规定,一个 LET 语句只能使用一个赋值号,把一个表达式的值赋给一个或 n 个变量,不允许写成

let A=2+3, B=3+5

也不允许写成

let C=D=E=A+B

3. 用 LET 语句能够对字符串变量值中的子串赋值,格式为

LET〈串变量〉[i : j]=〈字符串表达式〉

其作用是用〈字符串表达式〉的结果来取代〈串变量〉中第 i 个字符到第 j 个字符。若 i<1,系统自动按 1 处理;若 j 大于串长度,则按串长度处理;若 i>j,则将〈字符串表达式〉的结果插入〈串变量〉的第 i 个字符前,而且不删除任何字符。

如 let a$[1 : 1]="x"　　将 a$ 中的第 1 个字符改为"x"。

let a$[4 : maxnum]=""　　将 a$ 中从第 4 个字符起全删除。这里""表示一个空字符串。

let a$[1 : 0]="The ␣"　　在 a$ 的前头插入"The ␣"。

2.4.3 INPUT 语句

INPUT 语句称为键盘输入语句,它的功能是把由键盘输入的一组数据传送到内存中具有相应类型的变量中去。它的一般格式为:

INPUT [PROMPT〈提示字符串〉:]〈变量 1〉,〈变量 2〉,……

计算机执行到 INPUT 语句时，暂停程序运行，在背景窗口将 PROMPT 后面的提示字符串内容照原样显示出来，然后给出一个"?"号，等待用户按顺序输入与变量对应的数据；数据输完后打一个回车键，程序才继续执行。在数据输入过程中，各数据间应以逗号分隔。

例 2.4.1 输入 a,b 的值，求 $c=\sqrt{A^2+B^2}$。程序如下：

```
! Calculate squareroot of a&b.
!
input prompt"a,b is:" :a,b
let c=sqr(a * a+b * b)
print
print"squareroot is:"; c
end
```

```
Ok. run
a,b is: 3,4↙ (带下划线者表示由用户键入的数据)

squareroot is: 5
Ok.
```

INPUT 语句是 BASIC 语言提供的一种以人机会话的方式进行数据输入的手段。采用这种输入数据，提高了程序的灵活性，如例 2.3.1 中要求 $\sqrt{5^2+6^2}$ 或其它任意两个数的平方根时不需修改程序，只需重新运行一次程序便可。它特别适合于参数变化而运算方法不变、或者在程序设计时某些参数还难以确定的情况。下面对 INPUT 语句作进一步说明。

(1) 在 INPUT 语句中，"PROMPT〈提示字符串〉："是可选用的部分。当不使用短语"PROMPT〈提示字符串〉："时，程序执行完 INPUT 语句将显示一个问号，等待用户的回答。如例 2.4.1 可以修改为

```
! Calculating squareroot of a&b.
!
print"a,b";
input a,b
let c=sqr(a * a+b * b)
print
print"squareroot is:"; c
end
```

```
Ok. run
a,b? 3,4↙

squareroot is: 5
Ok.
```

(2) 在 INPUT 语句中所列出的变量，可以是数值变量，也可以是字符串变量。应向它们输入相应类型的常量。例如，

输 入 语 句	应输入的合法常量
input a	? 5
input a,b,c	? 8.3,−6.5,17.3
input t $. a,x	? san,−3,6
input b $,c $,a $	? wang,"3 sep 1988",56

回答字符串变量的值时，字符串可以用双引号括起来，也可以不用。当输入首部或末尾为空格，以及中间有逗号的字符串时，必须用双引号括起来。也可以输入空字符串""。

(3) 下面几种回答将会引起出错：

① 需要回答数字时，回答了一个字符串，将会引起"类型出错"的错误。若要求回答字符串时，回答了几个数字，不显示出错信息，将这几个数字不作为数值量而作为字符串被存到相应的串变量中去了。

② 回答的数字是一个溢出数值(超过计算机允许的范围)。

③ 输入的数字中有空格。

④ 回答的数据个数太多。如输入数据的个数不足时，True BASIC 继续等待输入其它数据，直到满足需要为止。

⑤ INPUT 语句不能进行运算，因此由键盘向变量传送的只能是常量，而不能是表达式或变量名。

例 2.4.2

```
! Input one number and stop.
!
input n
end
```

```
Ok. run
? Taiyuan↙(应输入数值，错输入了字符串)
String given instead of number. Please retry from bad item.
? le3000↙ (输入的数太大)
Overflow.  Please retry from bad item.
? 1,000↙ (输入数据个数太多)
Too many input items. Excess ignored.
Ok.
```

要特别注意，数据应在一行内输完，如果不能把所有数据写在一行，就要在一个输入行的最后加一个逗号，然后打入"回车"，这样 True BASIC 将在下一行再次给出"?"，用户便可以把剩下的数据继续输入。逗号表示后面还有数据要输入，因此千万不可在一个数据中间插入逗号(这样就会把一个数变成了两个数或多个数，增加了数据的个数)。

例 2.4.3

```
!
input A,B,C
```

```
print"Thanks. "
end
```

```
Ok. run
? 1,2,3,4↙ (键入的数据太多,程序正常执行,但给出如下信息)
Too many input items. Excess ignored.
Thanks.
Ok. run
? 1,2↙ (键入的数据太少,要求增加)
Too few input items. Please add more.
? 3↙
Thanks.
Ok. run
? 1,2↙ (用“,”表示键入未完,系统再给出一个“?”)
? 3↙
Thanks.
Ok.
```

(4) INPUT 语句末尾以逗号结尾时,输入的数据可以比变量多。多余的数据被暂存,并让后面的 INPUT 语句读入。这样可以减少用户输入行。如程序段

```
print"name, address";
input name $
input address  $
```

便可以在第一个 input 语句执行后,在一行中回答 name $ 与 address $ 的值。即输入

wang li, "3 Beijing Road, Shang hai"

2.4.4 DATA/READ 语句与 RESTORE 语句

1. DATA/READ 语句:DATA 语句称建立数据区语句。它的格式为

DATA〈数据 1〉,〈数据 2〉,…

程序中的 DATA 语句,被输入计算机后,系统就把各个 DATA 语句中的数据按它们书写的先后顺序,在内存的特定区中形成一个数据存储区。这个数据区中的数据次序与 DATA 语句的个数无关,只与它们的相对顺序有关。如语句:

```
……
data 9, 8, 7, 6, 5, 4, 3, 2, 1
```

与以下语句:

```
data 9…
……
data 8, 7, 6, 5
……
```

```
data 4,3
……
data 2,1
```

所建立的数据区中数据的存放情况是相同的。

DATA 语句的作用是在程序输入后，向数据区提供一些数据，供程序执行时使用。

READ 语句称为读数据区语句。它的功能是把数据区中已有的数据(它们是由 DATA 语句建立的)依次传送到一些变量中去。这些变量是由在 READ 语句中指定的。

READ 语句的格式如下：

READ〈变量 1〉，〈变量 2〉，…

程序执行到 READ 语句时，便会把数据区中的数据依次传送给 READ 语句中所指定的变量。或者说让 READ 语句中的各变量依次去"读"数据区中的数据。为了能表示 READ 语句中的变量读取数据区中的数据的情况，可以设想在数据区有一个"数据指针"。"数据指针"一开始指向数据区内的第一个数据，当有一个变量读取了指针指向的数据后，指针就自指向下一个数据。

例 2.4.4 一个同学期末考试考了 5 门课，为了求这 5 门课的平均成绩，可以把他们用 DATA 语句存在程序中，然后用 READ 语句读到变量中去，再进行计算。程序如下：

```
! Calculating average score of a student.
!
data 87, 89, 70, 76, 80
!
read scorel
read score2, score3, score4
read score5
let average_score=(score1+score2+score3+score4+socre5)/5
print"Average_score of the student is:"; average_score
end
```

```
Ok. run
Average_score of the student is:80.4
Ok.
```

图 2.4 为该程序执行过程中数据指针的变化情形示意图。图 2.4(a)为程序输入后，尚未执行 Read 语句前数据区状况。在执行了第一个 Read 语句后，score 1 读入数据 87，数据指针移到指向第二个数据的位置，见图 2.4(b)。执行完第二个 Read 语句(score2, score3, score4 读入数据 89,70,76)后数据指针的位置见图 2.4(c)所示。图 2.4(d)为 score5 读数后数据指针的位置。

下面对 DATA/READ 语句的使用做进一步说明：

(1) 在一个完整的程序段中，数据区可以由一个 data 语句建立，也可以由多个 data 语句建立，而且 data 语句在程序中的位置不影响程序的执行结果。data 语句可以集中地放在程序的首部或尾部(END 语句之前)，也可以分散地放在要读取这些数据的 READ 语句附

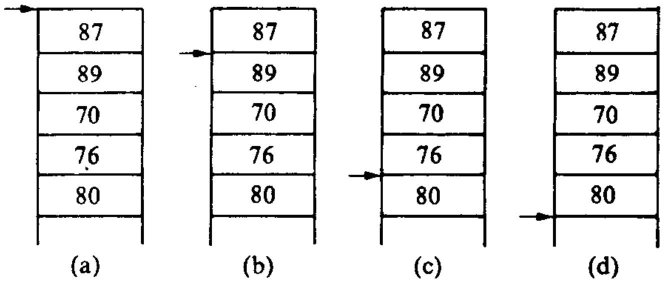


图 2.4

近。如例 2.4.4 也可以写为

```
! Calculating average score of a student.
!
read score1
data 87
data 89,70
read score2, score3, score4
data 76, 80
read score5
let average_score=(score1+score2+score3+score4+score5)/5
print"Average_score of the student is:"; average_score
end
```

```
Ok. run
Average_score of the student is: 80.4
Ok.
```

(2) READ 语句中的各变量,可以是数值变量,也可以是字符串变量。不同的变量要求 DATA 语句按顺序提供类型对应相同的数据,若类型不对,就会产生"wrong type"的错误。

(3) READ 语句的功能仅仅是传送,DATA 语句的作用仅仅是建立数据区,因此 DATA 语句所提供的数据必须是常量,而不能是表达式。READ 语句是执行语句,DATA 语句是非执行语句。

(4) 一个程序建立的数据区中的数据个数不能少于(可以多于)所有 READ 语句中的变量个数,否则便会出现"Reading past end of data,"(读出界)的错误。为了防止出现读出界的错误,True BASIC 提供了 MORE DATA 和 END DATA 两个函数来测试数据区是否还有数据可读。

如果数据区中还有数据可读,则函数 MORE DATA 的值为真,否则为假;如果数据区中无数据可读,则函数 END DATA 的值为真,否则为假。

例如例 2.4.4 的程序可以改为

```
! Calculating average score of a student.
!
data 87, 89, 70, 76, 80
```

```
!
let t,sum=0
do while more data
   read score
   let sum=sum+score
   let t=t+1
loop
print"Average_score of the student is:"; sum/t
end
```

```
Ok. run
Average_score of the student is:80.4
Ok.
```

do while more data——loop 结构的意思是,"当数据区还有数据可读时,便执行循环,在循环中用 read 语句读数据,然后将读入的数累加到 sum 中,若数据区中没有数据了,就不再循环。"END DATA 将告诉程序"数据区是否结束了"。

上面程序中的循环部分可改写为

```
do
  read score
  let sum=sum+score
  let t=t+1
loop until end data
```

2. 恢复数据区指针语句(RESTORE):程序建立的数据区中的数据一经读过,便不能再使用。若要重复使用数据区中的数据,必须借助 RESTORE 语句。RESTORE 语句的功能是将数据指针拨回到开始的位置,由于 Read 语句是读取数据指针所指向的数据,这样便可以重新使用数据区中的数据了。RESTORE 语句很简单,格式为

RESTORE

例 2.4.5 按下面的对应关系给变量赋值:

A	B	C	D	E	F	G	H	I	J	K	L
↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑
1	2	3	4	1	3	5	7	2	4	6	8

程序如下:

```
! A example with the RESTORE.
!
read A,B,C,D
restore
read E,X,F,X,G,X,H
restore
read X,I,X,J,X,K,X,L
```

```
data 1,2,3,4,5,6,7,8
end
```

在程序未执行时，指针位置如图 2.5(a)。在执行完第一个 READ 语句后，指针位置如图 2.5(b)。在执行完第一个 RESTORE 语句后指针位置如图 2.5(c)。执行完第二个 READ 语句后，指针位置如图 2.5(d)。执行完第二个 RESTORE 语句后，指针位置如图 2.5(e)。执行完第三个 READ 语句后，指针位置如图 2.5(f)。

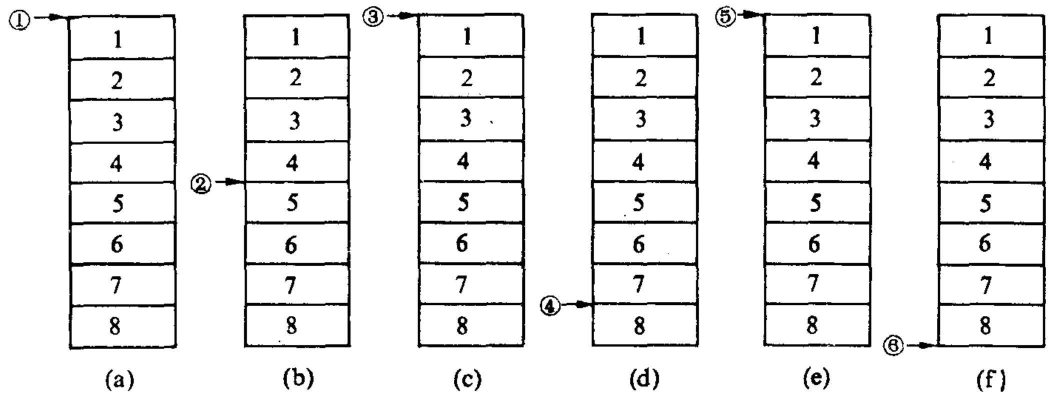


图 2.5

程序中的变量 x 是用来使下一个有用变量跳过一个数据来读数据，通常称为“假读”。请读者写出程序，执行后变量 A,B,C,D,E,F,G,H 的值。

2.4.5 数据传送语句小结

前面已经介绍了四种数据传送语句，小结如下：

1. 这四种数据传送语句中只有 PRINT 语句是向显示器和打印机传送数据的，是唯一的输出语句。

2. 在 True BASIC 中，只有 PRINT 语句与 LET 语句有运算功能。然而 PRINT 语句是把运算的结果直接输出，以后需要时还须重新运算。而 LET 语句是把计算的结果保存在内存变量中，以后需要时可以直接调用而无须重新计算。因此，若一个程序中要多次使用某一表达式的值时，应先用 LET 语句把它传送到一个变量中保存。

3. LET,INPUT,READ 都能够把数据传送到变量之中。要把一个表达式的值传送到一个或多个变量之中，只能用 LET 语句。然而一个 LET 语句只能给一个变量赋值或者给多个变量赋以同一个值。要给不同的变量传送不同的值需要用多个 LET 语句，也可以用一个 INPUT 或一个 READ 语句。

INPUT 语句与 READ 语句都能给多个变量传送不同的数据。但 READ 语句所传送的数据是由程序中的 DATA 语句给出的(有时称是保存在 DATA 语句中的)，而 INPUT 语句传送的数据是程序执行过程中由操作人员从键盘上输入的。

因此，如果数据是在程序设计时已经给定的，就应使用 LET 或 DATA/READ 语句(当数据量较大时，应使用 DATA/READ 语句)。这样可以提高执行速度。如果在程序设计时数据还不能确定，或考虑到数据会依不同的条件而变动，就应使用 INPUT 语句。但是机器执

行 INPUT 语句时，要等待操作人员从键盘上输入数据，所以程序执行速度是很低的。

这四种传送语句的作用如图 2.6 所示。

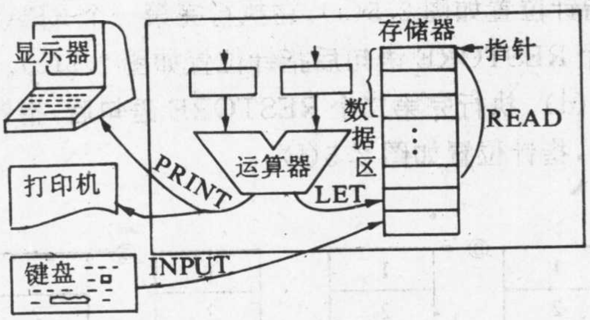


图 2.6

2.5 选取型程序结构

选取型程序结构是一种分支结构，根据给定条件是否满足，从两种或多种可能的操作方案中选择一种。如同到三岔路口一样，可以从两条路径中选择其中一条路径。

应当注意，选取是有条件的。在程序中，这个条件用逻辑表达式表示。

2.5.1 逻辑表达式

逻辑(Logic)学是一门有关推理和论证的科学，一般用于处理命题的“真”(true)或“假”(false)。在计算机程序中，一个逻辑表达式就是一个命题，它的值不是数值，而只能是“true”或“false”，故也称为二值逻辑。逻辑表达式就是程序进行选取的条件。

用关系运算符(＜、＞、＝、＜＞、＜＝、＞＝等)连接的数值表达式或字符表达式是关系表达式，用逻辑运算符将两个或多个关系表达式或逻辑常量(true 或 false)联接起来的式子是逻辑表达式。

1. 关系运算符和关系表达式：像 a＞3，a＜＞b，a＋b＜＝b＊t 等都是关系表达式。关系表达式中包含一个关系运算符。

表 2.8 列出 True BASIC 中使用的关系运算符。关系运算的优先级别是相同的，用圆括号可以指定某些运算优先。

可以对两个数值数据或字符串数据进行关系运算，但不允许对一个字符串表达式和一个数值表达式进行关系运算，例如 a＞＝b$ 是不合法的。

数值数据的比较是大家都已熟悉的，即按代数值进行比较。对字符串来说，它本无大小可言，但是人们往往需要对字符串(如人名、地名、字典中的字与词)进行排序，所以就规定了一些规则使之能够互相比较。通常用字符的 ASCII 码代表字符进行比较。如果一个字符的 ASCII 码值大于另一个字符的 ASCII 码值，就认为前者“大于”后者。ASCII 排列有一定的规则：数字字符比字母字符“小”，大写字母字符比小写字母字符“小”；同一字体的字母中，按英文字典顺序排在前面的字母字符比它后面的“小”；空格比一般字符都“小”。按照这些规则，不用翻看 ASCII 字符表也能知道常用字符的大小，如“A”＜“a”、“A”＜“B”、“9”＜“A”、“ ”＜“0”等。

表 2.8

关系运算符	含　义	示　例
<	小于	a<b 或 a $ <b $
<=或=<	小于等于	a<=b 或 a $ <=b $
=	等于	a=b 或 a $ =b $
>=或=>	大于等于	a>=b 或 a $ >=b $
>	大于	a>b 或 a $ >b $
<>或><	不等于	a<>b 或 a $ ><b $

两个字符串比较时，从组成这两个字符串的最左端第一个字符开始自左而右逐对一一对应地进行。如果两个字符串中的所有字符都一一对应相等，则称这两个字符串相等；在自左而右地比较中，如发现有一对不同的字符，便以这两个字符比较结果作为两个字符串比较的结果，其后的字符不必考虑。如"These">"That"，"Boy"<"Girl"等。这个规律与英文字典排列是一样的，即在字典中在后面的字比前面的字"大"。这个规律是很容易记住的。

应特别注意：空格也是一种字符。这一点往往初学者忽视。例如"ABC ␣ DEF"<"ABCDEF"。

还应指出：关系表达式的值是一个逻辑常量(true 或 false)而不是数值。

2. 逻辑运算和逻辑运算符

逻辑运算的对象是逻辑值。True BASIC 可以提供 NOT(非)、AND(与)和 OR(或)三种逻辑运算。像 a>0 AND a<S，a=3 OR a=5，NOT x>0 都是逻辑表达式。表 2.9 为三种逻辑运算的定义及示例。

表 2.9

逻辑运算	结　果	逻辑表达式示例
NOT true	false	NOT　2<5 * 2
NOT false	true	NOT　5<2+1
true AND true	true	1=1　AND　3<5+3.5
true AND false	false	"a"<>"b" AND 5<3
false AND false	false	1<1　AND　5<3.5
true OR true	true	1=1　OR　3<5
true OR false	true	"man">"boy"OR"book">"pen"
false OR false	false	1>1　OR　5<3

请注意：True BASIC 中没有逻辑常数，因此不能在表达式中出现"true"或"false"。例如，IF(a>0)=true then print a 这种写法是错误的。

三种逻辑运算的优先顺序依次为：NOT，AND 和 OR(不分大小写)。如

a $ ="Li" or not a $ ="Li" (结果为"真")

3. 逻辑函数

逻辑函数能送回一个逻辑值。如 2.3.3 中已经介绍过的函数 END DATA 和 MORE DATA 就是能送回数据区中数据是否已读完的两个逻辑函数。

2.5.2 IF 型结构控制

True BASIC 的分支结构有两种：一种是二路分支，一种是多路分支。本小节介绍的 IF 型选取结构就是一种二路分支结构。多路分支结构将在 2.5.3 节介绍。

在 ture BASIC 中，二路分支的条件是逻辑表达式，它有以下三种标准格式：

格式Ⅰ：

```
IF〈条件〉THEN〈语句 1〉[ELSE〈语句 2〉]
```

格式Ⅱ：

```
IF〈条件〉THEN
   ……
   ……  〈语句块 1〉
   ……
[ELSE
   ……
   ……  〈语句块 2〉
   ……]
END IF
```

格式Ⅲ：

```
IF〈条件 1〉THEN
   ……
   ……  〈语句块 1〉
   ……
ELSE IF〈条件 2〉THEN
   ……
   ……  〈语句块 2〉
   ……
ELSE IF〈条件 3〉THEN
   ⋮
[ELSE
   ……
   ……  〈语句块 n+1〉
   ……]
END IF
```

格式Ⅰ与格式Ⅱ都是单条件的，它们的结构都如图 2.7 所示。图中"T"和"F"分别代表"真"和"假"，代表条件"成立"或"不成立"。格式Ⅲ是多条件的，其结构如图 2.8 所示。这种结构形成了多层二分支结构。

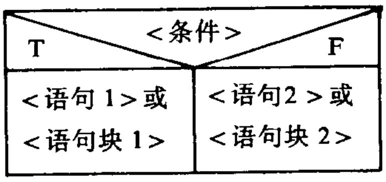


图 2.7

我们还注意到：格式Ⅰ是一种单行形式，所有内容都写在一行中。格式Ⅱ与格式Ⅲ都是多行结构，整个结构以 IF 语句开始以 END IF 语句结束，中间部分关键字 ELSE(或 ELSE IF)分隔成两个(或多个)语句块，书写成的程序段结构非常清晰。

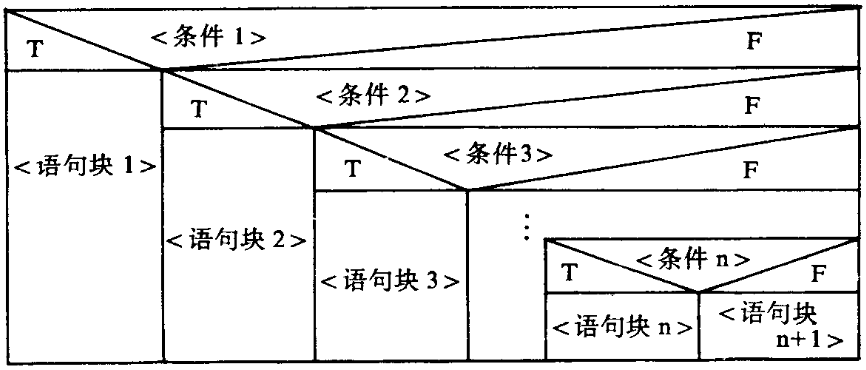


图 2.8

例 2.5.1 从键盘上输入三个数，打印出最大的一个数。

在例 1.3.1 中已经按逐步细化方法设计过这一问题的算法。这里再系统地介绍一下如何逐步细化地设计求解该题的程序。

(1) 对题目在“做什么”一级抽象可得如下算法：

S1：输入三个数；

S2：找最大的一个数；

S3：打印最大的数。

(2) 对上述算法细化得：

S1：INPUT A,B,C；

S2：又分为两步：

　S2.1：在 A 和 B 中选大者送入变量 MAX 中；

　S2.2：在 C 和 MAX 中选大者送入变量 MAX 中。

S3：PRINT MAX。

(3) 用 N-S 图描述上述算法如图 2.9 所示，这时的算法已经可以直接用 True BASIC 描述了。

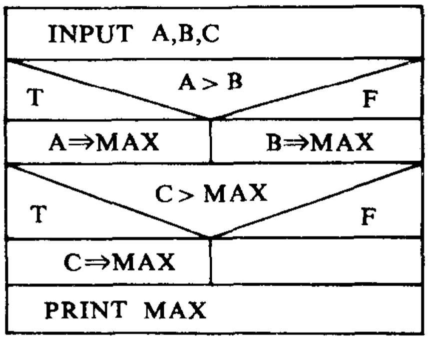


图 2.9

首先用单行 IF——THEN——ELSE 结构描述如下：

```
! Find the maximum from three numbers.
!
input prompt"a,b,c=" : a,b,c
if a>b then let max=a else let max=b
if c>max then let max=c
!
print"max="; max
end
```

```
Ok. run
a,b,c=2,9,5↙
max=9
Ok.
```

也可以用多行 IF——THEN——ELSE 结构的 True BASIC 程序来描述：

```
! Find the maximum from three numbers.
!
input prompt"a,b,c=" : a,b,c
if a>b then
      let max=a
   else
      let max=b
end if
if c>max then
```

```
    let max=c
end if
print "max=";max
end
```

```
Ok. run
a,b,c=2,8,5↙
max=8
Ok.
```

如果采用图 1.31(g)和(h)所示的算法，又可以得到下面的 True BASIC 程序：

```
! Find the maximum from three numbers.
!
input prompt"a,b,c=" : a,b,c
if a>b then
  if a>c then
    let max=a
  else
    let max=c
  end if
else
  if b>c then
    let max=b
  else
    let max=c
  end if
end if
print"max=";max
end
```

```
Ok. run
a,b,c=3,9,2↙
max=9
Ok.
```

这种 IF 结构中又含有 IF 的格式，称为 IF 嵌套(注意，有嵌套时采用缩格书写格式，即把嵌套的部分内缩 2～3 格，可以使程序更清晰、易读)。这时，可以很容易地把它改写为 ELSE IF 结构：

```
! Find the maximum from three_numbers.
!
input prompt"a,b,c=" : a,b,c
if a>b then
  if a>c then
```

```
    let max=a
  else
    let max=c
  ent if
else if b>c then
    let max=b
else
    let max=c
end if
print"max=";max
end
```

```
Ok. run
a,b,c=2,9,4↙
max=9
Ok.
```

请比较以上两个程序，它们的流程图都是图 1.31(h)，两者的逻辑是相同的，只是具体语句的写法不同。ELSE IF 结构的含义是："否则，如果……则……，"它实际上是将原来分写在两行上的 ELSE 和 IF 连起来写。也就是说，ELSE IF 结构是多行 IF——THEN——ELSE 结构中的 ELSE 分支中又嵌套一个 IF 结构时的一种变换形式。原先在 ELSE 分支中嵌套一个 IF 结构时，这个嵌套的 IF 结构要有 END IF 关键字，而改写成 ELSE IF 结构时，不要求 END IF 与 IF 配对。请仔细对照上面两个程序。如果嵌套层次比较多时，如果不用 ELSE IF 结构而用多行 IF——THEN——ELSE 结构，则每一层都有一个 END IF，而且用缩进格式表示嵌套可能会出现向右缩进到纸的右端，很不方便。而用 ELSE IF 结构可以省写若干行 END IF，一系列的 ELSE IF 只需要一个 END IF。在多分支选择时用 ELSE IF 结构是比较方便的，一目了然。例如，

```
if p<=100 then
  let r=0.03
else if p<=200 then
  let r=0.04
else if p<=300 then
  let r=0.05
else if p<=400 then
  let r=0.06
else
  let r=0.07
end if
```

其流程图也可以用图 2.10 形式表示。

例 2.5.2 求一元二次方程的实根。

(1) 算法设计：一元二次方程式 $ax^2+bx+c=0$ 的根决定于它的三个系数 a，b，c。所以

p 的值				
≤100	101～200	201～300	301～400	>400
r=3%	r=4%	r=5%	r=6%	r=7%

图 2.10

求一元二次方程式的根这一问题可以抽象地描述为如下算法：

S1：输入一元二次方程式的系数 a，b，c；

S2：求一元二次方程式的根；

S3：打印一元二次方程式的根。

上述算法中 S1 已经可以直接代之以 True BASIC 的语句了。下面对 S2 与 S3 细化。

由代数学可知，一元二次方程式：$ax^2+bx+c=0$ 的根为

$$x_{1,2}=\frac{-b\pm\sqrt{b^2-4ac}}{2a}$$

并且，在实数范围内，一元二次方程式的根决定于判别式：$\Delta=b^2-4ac$。$\Delta>0$，方程有两个不等的实根；$\Delta=0$，方程有两个相等的实根；$\Delta<0$，方程没有实根。Δ 与一元二次方程的根的关系也可以用图 2.11 所示的“判定表”来直观地描述。判定表中双线左上部为各种条件的罗列（Y 代表条件成立。N 代表条件不成立），左下部为各种可能结果的罗列。在每一列上可以看出在不同的条件组合时得到的结果。“√”表示“选中”某项结果。

根据图 2.11 所示的判定表，对 S2 细化，可以得到图 2.12 所示求一元二次方程根的算法。

Δ>0	Y	N	N
Δ=0	N	Y	N
Δ<0	N	N	Y
二个根	√		
一个根		√	
无实根			√

图 2.11

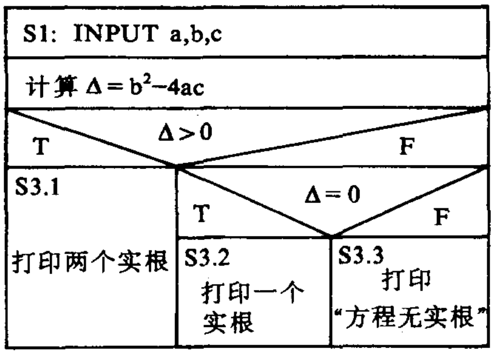


图 2.12

当 $\Delta>0$ 时，方程的两个不同的实根为

$$x_{1,2}=\frac{-b\pm\sqrt{\Delta}}{2a}$$

当 $\Delta=0$ 时，方程有两个相同的实根：

$$x_{1,2}=-b/(2a)$$

到此为止，似乎已可以按图 2.12 所示算法细化，直接写出 True BASIC 程序了。但是，还有一点没有考虑，即若给定系数 a=0 时，怎么办？这时方程只有一个实根

$$x=-(c/b)$$

所以还要对图 2.12 所示算法进行一些修改，得到图 2.13 所示算法。由这一修改可以看出，逐步细化不是绝对的。在逐步细化的过程中，发现原来的方案不够准确，有考虑不周之处或有错误之处时，应及时修改，调整原来的算法。

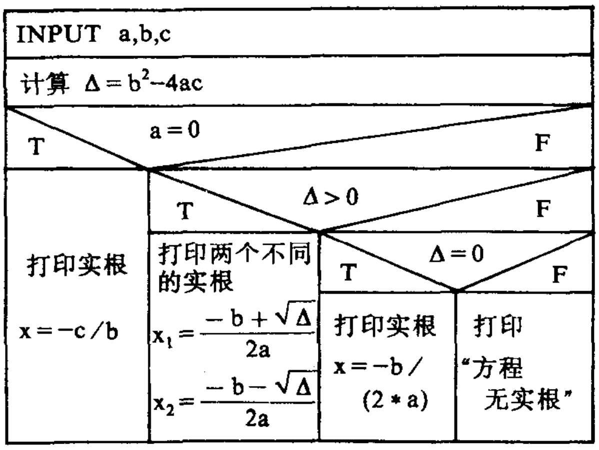


图 2.13

(2) 程序编码：对图 2.13 所示算法稍作修改，便可以用 True BASIC 描述了。程序如下：

```
! Solve quadratic equation, given coefficients.
!
print "Give coefficients";
input a,b,c                ! get the coefficients
let discr=b*b-4*a*c        ! compute discrimit
if a=0 then                ! if a=0
  print "Oen root at"      ! now it's really a linear equation
  print "x="; -(c/b)
else if discr>0 then
  let s=sqr(discr)
  print "The roots are:"
  print "x1=";(-b+s)/(2*a)
  print "x2=";(-b-s)/(2*a)
else if discr=0 then
  print "The root is:"
  print "x="; -b/(2*a)
else
  print "No real roots."
end if
end
```

```
Ok. run
Give coefficients? 2,3,5↙
No real roots.
```

```
Ok. run
Give coefficients? 1,4,4↙
The root is:
x=-2
Ok.run
Give coefficients? 1,4,3↙
The roots are:
x1=-1
x2=-3
Ok.
```

我们看到。当有多个 ELSE IF 结构中，程序总是按先后次序来检查每个 ELSE IF 后面的条件是否满足。

最后还要说明的一点：在所有的 IF 型选取结构中，ELSE 部分都是可选的（即可有可无的），用户可以根据需要决定是否要加上 ELSE 部分。对于三种标准格式，没有 ELSE 部分的三种格式如下：

```
IF〈条件〉THEN〈语句〉
```

```
IF〈条件〉THEN
    ......
    ......  }〈语句块〉
    ......
END IF
```

```
IF〈条件 1〉THEN
    ......
    ......  }〈语句块 1〉
    ......
ELSE IF〈条件 2〉THEN
    ......
    ......  }〈语句块 2〉
    ......
ELSE IF〈条件 3〉THEN
    ......
    ......  }〈语句块 3〉
    ......
ENDIF
```

如例 2.5.2 的程序可以改写为：

```
! Solve quadratic equation, given coefficients.
!
print "Give coefficients";
input a,b,c
let discr=b*b-4*a*c
if a=0 then
  print "One root at:"
  print "x="; -(c/b)
else if discr>0 then
  let s=sqr(discr)
  print "The roots are:"
```

```
  print "x1=";(-b+s)/(2*a)
  print "x2=";(-b-s)/(2*a)
else if discr=0 then
  print "The root is:"
  print "x1,x2="; -b/(2*a)
else if discr<0 then
  print "No real roots."
end if
end
```

```
Ok. run
Give coefficients? 2,3,4↙
No real roots.
Ok. run
Give coefficients? 1,4,4↙
The root is:
x1,x2=-2
Ok. run
Give coefficients? 1,4,3↙
The roots are:
x1=-1
x2=-3
Ok.
```

但是这种程序结构所描述的逻辑关系不如有 ELSE 时清晰，应尽量不用。

2.5.3 CASE 型结构控制

CASE 型选取结构是一种多路分支结构。在 True BASIC 中，它是用 SELECT CASE 语句实现的。SELECT CASE 的格式如左：

下面对 SELECT CASE 语句作几点说明：

(1) SELECT CASE 语句的功能：SELECT CASE 语句是 IF 语句的更一般化的形式，它根据测试表达式的值，在顺序排列的 CASE 表中选择执行一个语句块。其逻辑结构如图 2.14 所示 N-S 图。它的执行过程是：在进入 SELECT CASE 结构之前，程序应先给定测试表达式的值，进入 SELECT CASE 后，先由第一个 CASE 后的测试内容测试(有多个测试时，以“，”分隔，并按排列顺序进行)，一旦测试的条件满足，便执行该 CASE 后面的语句块内容；若一个 CASE 的全部测试都不满足，便进行下一个 CASE 的测试。若全部 CASE 都测试不满足，便执行 CASE ELSE 后的语句块。不论执行了哪一个语句块，最后都经 END SELECT 出口。所以，不管 SELECT CASE 内部结构多么复杂，它仍是一个单入口、单出口的结构。

```
SELECT CASE 〈测试表达式〉
  CASE 〈测试 11〉,〈测试 12〉,…
    ……
    ……  }〈语句块 1〉
    ……
  CASE 〈测试 21〉,〈测试 22〉,…
    ……
    ……  }〈语句块 2〉
    ……
      ⋮
  CASE ELSE
    ……
    ……  }〈语句块 n〉
    ……
END SELECT
```

〈测 试 表 达 式〉			
〈测试 11〉,〈测试 12〉,……	〈测试 21〉,〈测试 22〉,……	……	其它情形
〈语句块 1〉	〈语句块 2〉	……	〈语句块 n〉

图 2.14

(2) 测试表达式:测试表达式可以是字符串表达式,也可以是数值表达式。若是字符串表达式,后面所有的测试内容都应是字符串型;若是数值表达式,后面所有的测试内容都应是数值型的。

测试表达式应在进入 SELECT CASE 结构以前被赋值。

(3) 测试内容:测试只是采用常量,不允许采用表达式。它有三种可能的格式:

① 常量:如 0,1,2 或"A","B"等;

② 〈低值〉to〈高值〉,如 −20 to 20,"A" to "K"等;

③ is 〈关系操作符〉〈常数〉,如 is<5 等。

下面举例说明 SELECT CASE 语句的用法。

例 2.5.3 采用 CASE 型结构求一元二次方程根的程序。

(1) 算法设计:根据图 2.11 所示判定表很容易得到图 2.15 所示算法。

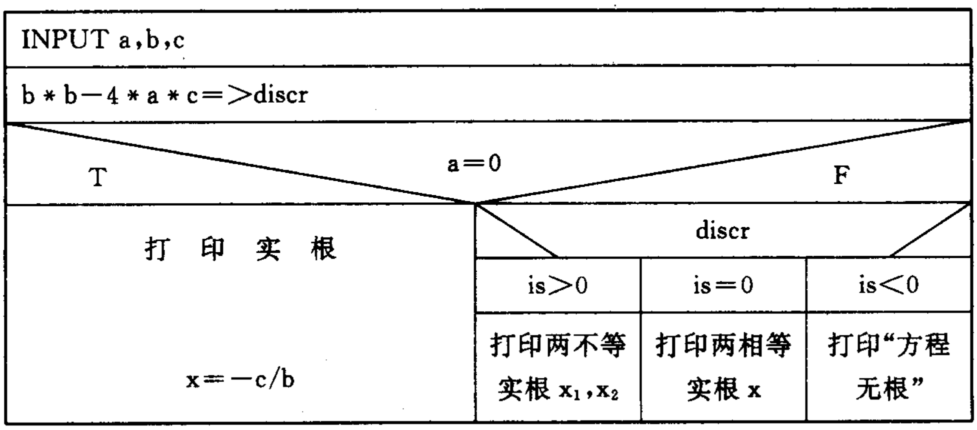


图 2.15

(2) 程序编码:根据图 2.15 所示算法,可以写出如下 True BASIC 程序:

```
! Solve quadratic equation, given coefficients.
!
print "Give coefficients";
input a,b,c                   ! get the coefficients
let discr=b * b-4 * a * c     ! compute discriminat
if a=0 then
  print "One root at:"        ! if a=0
  print "x="; -(c/b)          ! now it's really a linear equation
else
  select case discr
    case is>0
    let s=sqr(discr)
    print"x1=";(-b+s)/(2 * a)
    print"x2=";(-b-s)/(2 * a)
  case is=0
    print "The root is:"
    print "x="; -b/(2 * a)
  case else
    print "No real roots."
  end select
end if
end
```

这里采用了第③种方式进行测试。我们还注意到了,SELECT CASE 的功能与 IF——THEN——ELSE IF 极为相似,在许多情形下可以互相替代。

例 2.5.4 报数游戏:甲、乙、丙、丁、戊、己、庚、辛 8 人站成一排,按图 2.16 所示方法从 1 开始报数。问谁报到 19880426?

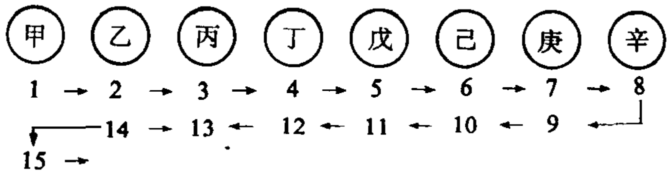


图 2.16

(1) 算法设计

由图 2.16 可以看出,这种报数从甲开始再回到乙,共报了 14 个数。所以按这种规律报数时,每报 14 个数是一个周期。而 19880426 便是某个周期中的一个数。我们关心的是 19880426 是这个周期中的第几个数,而与前面的周期无关。根据这一分析可以得到如下算法:

S1: 19880426⇒x;

S2：求 x 是报数周期中的第几个数；

S3：根据 x 是报数周期中的第几个数，确定这个数是谁报的。

S1 已无须细化。下面先对 S2 细化。要确定 x 是报数周期中的第几个数，只须用 14 对它求模便可知道，即进行运算：N＝MOD(X,14)。再对 S3 细化。进入含 x 的周期后，N 一定是 1～14 中的一个数，由图 2.16 所示关系，很容易得到图 2.17 所示判定表。此判定表就是对 S3 的一种描述，由它很容易得到 S3 的算法结构。图 2.18 是经细化得出的整个算法的结构。

N的值	1	Y	N	N	N	N	N	N	N
	2,0	N	Y	N	N	N	N	N	N
	3,13	N	N	Y	N	N	N	N	N
	4,12	N	N	N	Y	N	N	N	N
	5,11	N	N	N	N	Y	N	N	N
	6,10	N	N	N	N	N	Y	N	N
	7,9	N	N	N	N	N	N	Y	N
	8	N	N	N	N	N	N	N	Y
报X的人	甲	X							
	乙		X						
	丙			X					
	丁				X				
	戊					X			
	己						X		
	庚							X	
	辛								X

图 2.17

(2) 程序编码

根据图 2.18 所给出的算法，可以得到下面的 True BASIC 程序：

```
! A game: Count off.
!
let x=19880426
let n=mod(x,14)
select case n
   case 1
      print"Jia"
   case 2,0
      print"Yi"
   case 3,13
      print "Bing"
   case 4,12
      print "Ding"
   case 5,11
      print "Wu"
   case 6,10
```

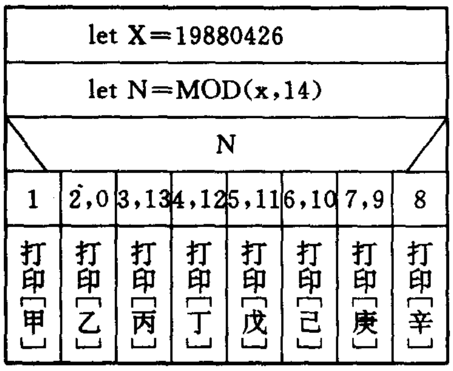


图 2.18

```
    print "Ji"
  case 7,9
    print "Geng"
  case 8
    print "Xin"
end select
end
```

与 IF 结构中的 ELSE 一样，SELECT CASE 结构中的 CASE ELSE 也是可选的。

2.6 循环型程序结构

循环结构也称重复型结构。它的功能是使一段程序能重复执行。被重复执行的部分称为循环体(body of loop)。

重复一般是有条件的。即在满足一定条件下才执行循环体(有条件的进入循环)，或者满足一定条件就不再执行循环体(有条件的退出循环)。或者进入和退出都是有一定条件的。重复控制结构的功能就是决定在什么条件下进入或退出循环。按照控制方式，True BASIC 的循环结构可分为 FOR 循环和 DO 循环两种类型。

2.6.1 DO 循环

1. 最简单的 DO 循环格式

最简单的 DO 循环格式如下：这种 DO 循环结构是没有条件的。

```
DO
    ......
    ......  〉〈循环体〉
    ......
LOOP
```

例 2.6.1 累加器程序：累加器程序是把由键盘键入的各数据累加，如同使用电子计算机进行累加一样。在运动会上累计得分或商店累计销售金额等都可以使用此程序。

程序如下：

```
! Program of accumulation.
!
let sum=0                        ! Get initial value.
do
  input prompt "x=":x            ! Get a value of x.
  let sum=sum+x                  ! Accumulate.
  print "sum is:"; sum           ! Print the sum.
loop
end
```

```
Ok. run
x=2↙
```

```
sum is: 2
x=3↙
sum is: 5
x=5↙
sum is: 10
x=8↙
sum is: 18
x=7↙
sum is: 25
x=(按[Ctrl]、[Break]键)
program stopped.
```

这个程序只要一启动,便会不断执行:要求用户由键盘输入 x 的值——累加——打印——再要求用户输入 x 的值……。这是一个无限循环。要让程序中止运行,可以按键盘上的暂停键[Ctrl]+[Break]。程序清单中的"■"表示程序中断后,光标指出的中断位置。

2. EXIT DO 语句

一般说来,循环是有条件的,即用户常常要求程序能在一定的条件下,结束循环,这时可使用 EXIT DO 语句。EXIT DO 语句能使控制转到循环的外部,它常常要嵌在 IF 语句中使用。

例 2.6.2 累乘器程序:累乘器程序能将输入的各数据相乘。程序如下,请读者比较它与累加器程序的异同。

```
! Program of re_multiply.
!
let m=1                       ! Get initial value
do
    input prompt "x=": x      ! Get a value of the x
    if x=0 then exit do       ! Exit of the loop
    let m=m * x
  print "m="; m
loop
end
```

```
Ok. run
x=2↙
m=2
x=3↙
m=6
x=5↙
m=30
x=4↙
m=120
x=0
```

Ok.

Ok.

我们注意到累乘器中 m 的初值为 1，因为如果使 m＝0，则所有累乘的结果就会为零。同理，输入的 x 也不应该等于零。所以规定一旦输入了 x＝0，便使流程转出循环，停止累乘。这里“x＝0”，是程序设计者自己指定的“循环结束标志”。当用户想要结束程序运行时，可给 x 输入一个“0”。请读者考虑，累加器程序中为什么要给 Sum 赋值“0”？而不能赋以“－1”或其它的数？

3. WHILE 型和 UNTIL 型

除 EXIT DO 以外，还可以使用 WHILE〈条件〉和 UNTIL〈条件〉来控制循环。True BASIC 提供了多种灵活的循环结构形式。WHILE〈条件〉和 UNTIL〈条件〉可以放在 DO 语句中，也可以放在 Loop 语句中，使用十分方便。

格式 1：

```
DO UNTIL〈条件〉
    ......
    ......  }〈循环体〉
    ......
LOOP
```

格式 2：

```
DO WHILE〈条件〉
    ......
    ......  }〈循环体〉
    ......
LOOP
```

格式 3：

```
DO
    ......
    ......  }〈循环体〉
    ......
LOOP UNTIL〈条件〉
```

格式 4：

```
DO
    ......
    ......  }〈循环体〉
    ......
LOOP WHILE〈条件〉
```

格式 5：

```
DO WHILE〈条件 1〉
    ......
    ......  }〈循环体〉
    ......
LOOP UNTILE〈条件 2〉
```

格式 6：

```
DO UNTIL〈条件 1〉
    ......
    ......  }〈循环体〉
    ......
LOOP WHILE〈条件 2〉
```

格式 1～格式 6 的流程图如图 2.19(a)、(b)、(c)、(d)、(e)、(f)。请注意图中菱形框出口处“真”和“假”的位置。

下面对 WHILE 型与 UNTIL 型的 DO 循环作如下几点说明：

(1) “条件”是任意的逻辑表达式，如 DO UNTIL a>b AND c>d。

(2) WHILE 型是当条件为真时，循环体被重复执行；UNTIL 型是当条件为真时，就停止执行循环体。因此要把 WHILE 型改为 UNTIL 型或改 UNTIL 型为 WHILE 型时，条件要变反。例如，“DO WHILE x<>0”和“DO UNTIL x＝0”作用相同。

例 2.6.3 假定在银行中存款 10000 元，按 11.25％的利率，一年后连本带利将变为 11125 元。你若将此款继续存入银行，试问多长时间就会连本带利翻一番？

首先用 DO UNTIL 型循环，程序如下：

```
! Compute interest on a bank account.
! Stop when the money has doubled.
```

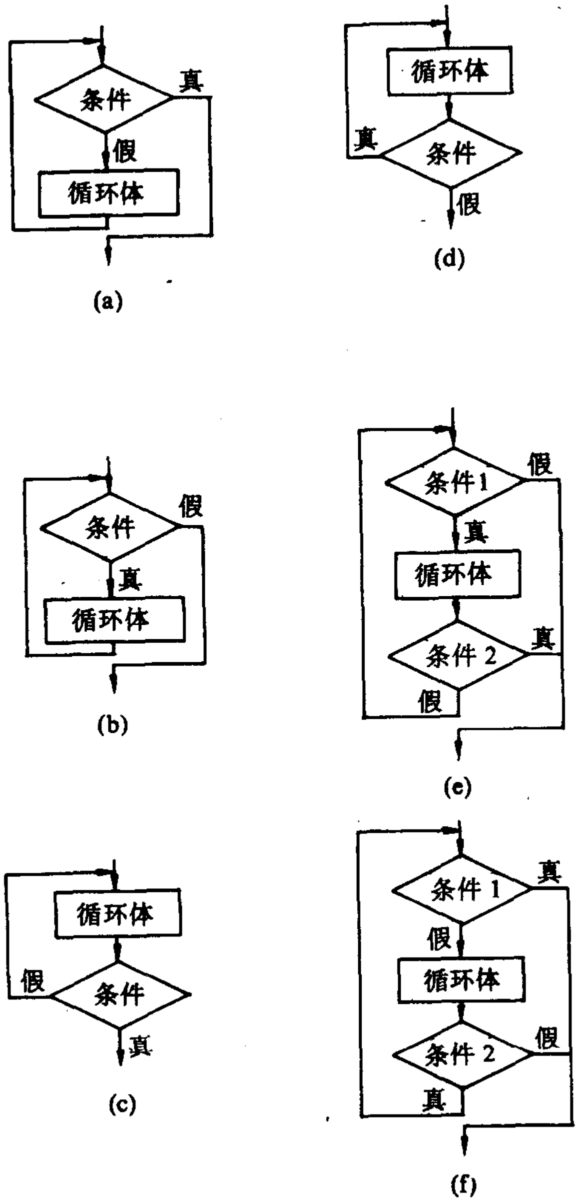


图 2.19

```
!
let money=10000                          ! Start with 10,000
let original=money                       ! Remember original amount
let interest=11.25/100                   ! Interest is 11.25% now
print "year","money"
do until money>=2 * original             ! Loop until money doubles
   print years,money
   let years=years+1
   let money=money+(interest * money)    ! Add in interest
loop
print "In"; years; "years,you'll have"; money
end
```

```
year      money
0         10000
1         11125
2         12376.6
3         13768.9
4         15317.9
5         17041.2
6         18958.3
In 7 years, you'll have 21091.1
Ok.
```

下面使用 DO WHILE 结构，程序如下：

```
! Compute interest on a bank account.
! Stop when the money has doubled.
!
let money=10000                 ! Start with 10,000
let original=money              ! Remember original amount
let interest=11.25/100          ! Interest is 11.25% now
print "years","money"
do while money<2 * original
   print years,money
   let years=years+1
   let money=money+(interest * money)
loop
print "in";years;"years,you'll have"; money
end
```

注意，这里条件已改为“money<2 * original”。若条件仍为“money>=2 * original”，将会得出错误的结果。请读者自己分析。

(3) WHILE〈条件〉或 UNTIL〈条件〉在 DO 语句中和在 LOOP 语句中，程序执行的情形是不相同的。在 DO 语句中时，在进入循环前，DO 就要对条件先进行测试，所以有可能因条件不满足而一次也不执行循环体。而在 LOOP 语句中时，则要先执行一次循环体，然后测试条件，以决定是否要再执行一次循环体。

例 2.6.4 求两个正整数 m，n(m>n)的最大公约数。

我们采用图 1.28(b)所示的欧几里德算法，可得到下面的程序：

```
! Program of Euclid.
!
!       This program calculates the greatest common
! divisor of two integer numbers using euclidean
! algorithm.
!
```

```
print "Get two integer numbers:";
input m,n
do
  let r=mod(m,n)
  let m=n
  let n=r
loop until r=0
print "The greatest common divisor is:";
print m
end
```

```
Ok. run
Get two integer numbers: ? 18,30↙
The greatest common divisor is : 6
Ok.
  Ok.
```

这个程序基本上是正确的。但若把它改成下面的DO UNTIL型循环时，情况就不同了。

```
! Program of Euclid.
!
!      This program calculates the greatest common
! divisor of two integer numbers using euclidean
! algorithm.
!
print "Get two integer numbers";
input m,n
do until r=0
   let r=mod(m,n)
   let m=n
   let n=r
loop
print "The greatest common divisor is:";
print m
end
```

```
Ok. run
Get two integer numbers? 12,30↙
The greatest common divisor is: 12
Ok. run
Get two integer numbers? 56,72↙
The greatest common divisor is: 56
Ok.
```

这后一个程序就是一个错误的程序了。由于“UNTIL r=0”是在DO语句中的，所以要

先检查。然而，程序中未对 r 赋过值，而 BASIC 在程序运行前使全部变量的初值为零，因此，循环体根本不会被执行。为了使程序能正确执行，应在 DO 语句之前使 r 具有一个值，以便进入循环并且要对程序做一些调整，见下面的程序（请注意带有标志“★”的语句）。

```
  ! Program of Euclid.
  !
  !      This program calculates the greatest common
  ! divisor of two integer numbers using euclidean
  ! algorithm.
  !
  print "Get two integer numbers";
  input m,n
★ let r=mod(m,n)
  do until r=0
★     let m=n
★     let n=r
★     let r=mod(m,n)
  loop
  print "The greatest common divisor is:";
★ print n
  end
```

```
Ok. run
Get two integer numbers? 21,56↙
The greatest common divisor is: 7
Ok. run
Get two integer numbers? 15,25↙
The greatest common divisor is: 5
Ok. run
Get two integer numbers? 0,5↙
The greatest common divisor is: 5
Ok. run
Get two integer numbers? 5,0↙
MOD and REMAINDER can't have 0 as 2nd argument.
```

这样就基本正确了。说它基本正确，是因为还没有考虑输入 n 为零这样一种特殊情形（请看程序最后一行的执行结果）。所以还需要给程序中加入一个条件，即 $n \neq 0$。

```
! Program of Euclid.
!
!      This program calculates the greatest common
! divisor of two integer numbers using euclidean
! algorithm
!
```

```
print "Get two iuteger numbers";
input m,n
if n<>0 then
  let r=mod(m,n)
  do until r=0
     let m=n
     let n=r
     let r=mod(m,n)
  loop
  print "The greatest common divisor is:";
  print n
else
  print "The divisor is 0,no solution."
end if
end
```

```
Ok. run
Get two integer numbers ? 5,0↙
The divisor is 0, no solution.
Ok.
```

这样，程序就完全正确了。它与前面一个程序的区别是不管在什么情况下都能正确地结束。也可以说程序的“坚固性”好。

上述几个程序比较简单，即使不画流程图也能直接编写出程序。对于复杂的程序，就必须在编写程序之前用某种方式写出详细算法。可以用来描述算法的工具很多。其中有一种非常灵活的算法描述工具称伪码(pseudo code)。它用自然语言描述操作，同时采用高级语言中的控制结构，它没有严格的语法规则，使用灵活方便。应当说明，伪代码不是写给计算机看的，计算机不能识别它。它和 N-S 图一样，只是人们设计算法时使用的一种辅助工具。任何一个有初步程序设计知识的人均可以方便地写出用伪码表示的算法。例如对上述求两个正整数的最大公约数的算法可以写为下面的伪码形式。

```
输入两个正整数 m,n
if n≠0 then
    求余
    do until 余=0
        互换
        再求余
    loop
    打印最大公约数
else
    打印“无解”
end if
end
```

伪代码程序中可以使用英文，也可以使用中文。对我国的广大程序人员来说，使用汉字更好。

这个伪码程序的每一步都可以用高级语言的语句代替，因此，根据这个伪代码程序很容易得到一高级语言源程序。

(4) 循环体内应当至少有一个能影响测试结果的操作，以便循环体经有限次重复后最终能退出循环。如例 2.5.2 中的 input 语句，例 2.5.3 中的 let money＝money＋(interest * money)语句，例 2.6.4 中的 let r＝mod(m,n)语句等。没有这些语句，循环将永远不会停止。

4. DO 循环嵌套：一个 DO 循环内还可以包含另外的 DO 循环，这种形式称为 DO 循环嵌套。

例 2.6.5 求 3～100 之间的素数。

(1) 算法设计

首先，可以很自然地把问题描述为图 2.20(a)，并将它细化得到图 2.20(b)。下面要对其中的 S1 与 S2 再细化。

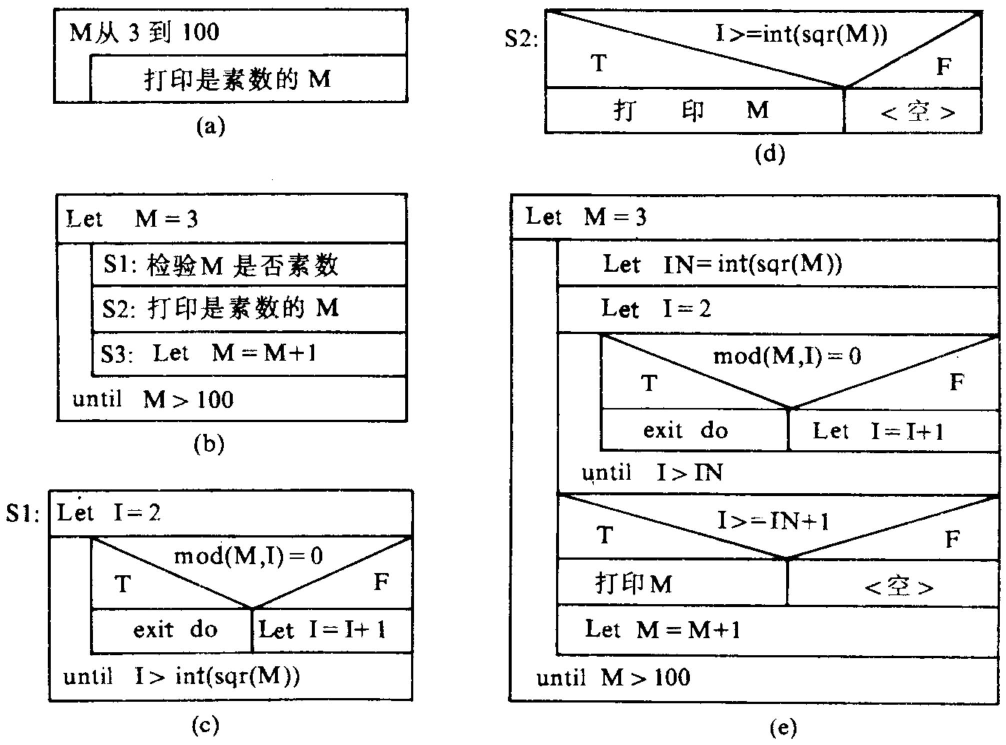


图 2.20

S1 是检验 M 是否素数。判别素数的方法很多。在例 1.3.2 中介绍了 Eratosthenes 筛法。现在用另一种方法：用 M 被 2 到 M－1 各整数除，如果都除不尽，则 M 是素数。可以这样做：以变量 I 为除数，它的值由 2 变到 M－1(每次 I 的值加 1)。每次判断 M/I 的余数是否为零，如果每次都不为零，则 M 必为素数。事实上，I 不必由 2 变到 M－1，只需变到 $\sqrt{M}$即可(如果 $\sqrt{M}$不是整数，则取整数)，例如，判断 17 是否素数，不必使 17 被 2 到 16 除，只需被 2 到 $\sqrt{17}$(即 2,3,4)除即可。因为一个数 M 如能被 A 整除，其商为 B，即 M＝A×B，A 和 B 之中必定有一个大于 $\sqrt{M}$，一个小于 $\sqrt{M}$。因此，在 2～ $\sqrt{M}$范围内必定可以找到一个除数

I,使 M 被 I 整除。

S2 的作用是打印是素数的 M。首先碰到的问题是,S2 在打印 M 之前如何知道 M 是素数呢?分析图 2.20(c)可知,当 M 是素数时,执行完 S1 后,I 的值应为 int(sqr(M))+1,若执行完 S1 后,I<int(sqr(M))+1,M 便不是素数。据此可以判定在执行 S1 之后,那个 M 是素数。S2 的算法如图 2.20(d)所示。

到此为止,算法中的每一步操作都是计算机可以执行的基本操作了。算法细化也告结束,经调整最后得到图 2.20(e)所示的算法。为了减少重复计算工作量,先将 int(sqr(M))算出来,存放在变量 IN 中。这时便可以编码了。

(2) 程序编码

由图 2.20(e)所示算法可以得到如下 True BASIC 程序:

```
! Look for prime numbers from 3～100
!
let m=3
do
let in=int(sqr(m))
let i=2
do
  if mod(m,i)=0 then exit do else let i=i+1
loop until i>in
if i>=in+1 then print m,
let m=m+1
loop until m>100
end
```

```
Ok. run
  3       5       7      11      13
 17      19      23      29      31
 37      41      43      47      53
 59      61      67      71      73
 79      83      89      97
Ok.
```

据此可将 S1 细化为图 2.20(c)。即先使 I 的初值为 2,判 M/I 的余数是否为零。如果为零则 M 不是素数。如余数不为零,使 I 加 1,再判 M 能否被 I 整除,如此循环,直到 I> $\sqrt{M}$ 为止。

这是一个 DO 循环嵌套结构。采用缩格书写格式,可以使程序中的层次清晰易读。

2.6.2 FOR 循环

DO 循环是用指定的条件是否满足来控制循环的。有时,事先已知道应循环的次数,可以简便地使用 FOR——NEXT 循环。

1. FOR 循环的格式:它的一般格式是

```
FOR 〈循环变量〉=〈初值〉 TO 〈终值〉 STEP 〈步长〉
   ……
   ……  〉〈循环体〉
   ……
NEXT 〈循环变量〉
```

先举一例说明其含义。

例 2.6.6 打印数字 1 到 10 的 True BASIC 程序。

```
! print numbers 1 through 10
!
for i=1 to 10 step 1
    print i;
next i
end
```

执行结果如下：

```
Ok.run
␣1␣␣2␣␣3␣␣4␣␣5␣␣6␣␣7␣␣8␣␣9␣␣10
Ok.
```

程序中的 i 就是循环变量，它的初值为 1，终值为 10，步长为 1。其执行过程见图 2.21。

2. 几点说明：

(1) FOR 循环通过循环变量与其终值的比较来控制循环过程。现以例 2.6.6 为例说明 FOR 循环的执行过程。

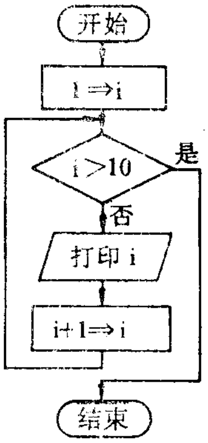


图 2.21

① 进入 FOR 循环时，先将循环初值 1 赋给循环变量(例 2.6.6 中为 I)，然后与终值(10)比较，若初值小于或等于终值，则进入循环。

② 进入循环后，执行完循环体，在 NEXT 语句中对循环变量进行修正，使循环变量的新值=循环变量原值+步长(例 2.5.6 中为"I=I+1")。

③ 执行完 NEXT 后，再将循环变量的现行值与终值比较，若循环变量大于终值，则跳出循环，否则继续执行循环体，并由 NEXT 语句使循环变量增值。如此重复，直到循环变量超过终值为止。

例 2.6.6 中的循环执行过程中 i 值的变化如表 2.10 所示。

(2) 可以看到：FOR 循环能自动对循环变量更值，每执行完一次循环体，NEXT 语句便对循环变量按步长作一次修正。所以 FOR 循环的循环体内不应再有给循环变量的赋值操作，以免破坏循环过程的正常过程。这是 FOR 循环与 DO 循环的一点区别。如例 2.6.6 的程序改为

```
for i=1 to 10 step 1
    let i=i+3
    print i;
```

```
next i
end
```

表 2.10

第n次	i值	与终值相比	执行循环体	第n次	i值	与终值相比	执行循环体
1	1	<10	执行循环	7	7	<10	循环
2	2	<10	循环	8	8	<10	循环
3	3	<10	循环	9	9	<10	循环
4	4	<10	循环	10	10	=10	循环
5	5	<10	循环	11	11	>10	退出
6	6	<10	循环				

则其打印结果将变为：

```
Ok. run
  4    8    12
Ok.
```

每次执行判断框时i的值如表2.11所示。

表 2.11

第n次	i值	与终值相比	执行循环体	第n次	i值	与终值相比	执行循环体
1	1	<10	执行循环	3	9	<10	循环
2	5	<10	循环	4	13	>10	退出

(3) 在FOR循环中，循环变量可以在循环体内引用，也可以不被引用。前者称为"操作型循环变量"，简称操作循变(参加操作但不应改变自身的值)。如例2.6.6中的循环变量i就是操作循变，它被"print i"语句引用。后者称为"形式型循环变量"，简称形式循变(不参加操作)。例如：

```
let x=1
for i=1 to 3 step 1
   print x;
next i
end
Ok. run
␣1␣␣1␣␣1
```

此处的i就是形式循变，它不在循环体内出现，没有被引用。此时的循环变量的作用只是控制循环的次数。为什么会执行三次循环体呢？这是由FOR语句决定的。

形式循变的名字不影响循环执行过程。如将例中形式循变名i改为j，即

```
for j=1 to 3 step 1
   print x
next j
```

作用仍同前。

(4) FOR 语句的作用是:①给循环变量赋以初值;②将初值与终值比较,以判断是否执行循环体;③将终值和步长的值记下来,以便以后执行 next 语句时对循环变量增值和比较。请注意,执行一个循环过程中只执行一次 FOR 语句,给循环变量只赋一次初值。千万不要以为执行一个循环过程会执行多次 FOR 语句,多次对循环变量赋值。

例 2.6.6 的正确流程图为图 2.21,如果将它的执行过程理解为图 2.22 是错误的。因为那样就会使 i 的值每次都等于 1 而不能正确实现循环。

(5) 当循环结束时,循环变量的当前值就是它最后得到的值。也就是第一次出现"超过"终值时的值。例如,在例 2.6.6 中在 next i 语句后面增加一个"print i"语句,如

```
for i=1 to 10 step 1
    print i
next i
print i
```

由于 i 的值为 11 时,超过了终值 10,才不执行循环体,因此,循环外面的 print 语句输出 i 的值应为 11。

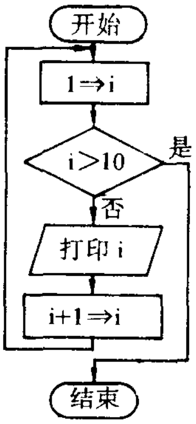


图 2.22

如果 for 语句改为

```
for i=1 to 10 step 3
```

其它语句不变,则结束循环后 i 的值为 13(i 的值变化情况为 1⇒4⇒7⇒10⇒13)。

如果初值超过终值,循环体一次也不执行。则循环变量的值保持为初值。例如,

```
for i=10 to 1 step 1
    print i
    next i
```

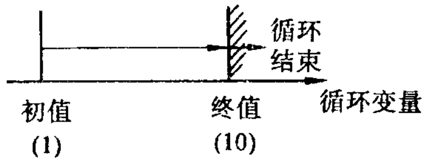
由于 i 的初值为 10,超过了终值 1,因此一次也不执行循环体,运行结果为

```
Ok. run
␣10
```

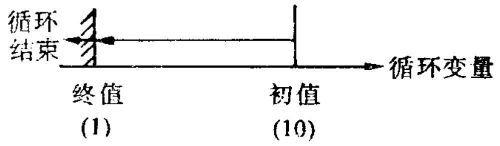
(6) 步长可以为正,也可以为负;可以是整数,也可以是小数。应当注意,循环终止条件的准确说法应当是:沿循环变量变化(初值到终值)的方向,循环变量的值"超过"终值(不是等于终值)。当步长为正时,"超过"终值的含义是"大于终值"。当步长为负时,"超过终值"的含义是"小于终值"。见图 2.23 所示。

为了能使循环体最少执行一次,对正步长,应使终值大于初值,如图 2.23(a)所示;对负步长,应使初值大于终值,如图 2.23(b)所示。下面是将例 2.6.6 中的步长改为-1 后的程序及执行结果:

```
! Print numbers from 10 down 1(backwards).
!
for i=10 to 1 step-1
    print i;
```



(a) 步长为正



(b) 步长为负

图 2.23

```
next i
end
```

```
Ok. run
  10    9    8    7    6    5    4    3    2    1
Ok.
```

步长是可选的，当不指出步长值时，隐含值为“1”。

3. FOR 循环嵌套：一个 FOR 循环内含有另一个 FOR 循环，称为 FOR 循环嵌套。

例 2.6.7 打印九九乘法表。程序如下：

```
! Print a multiplication table.
!
for a=1 to 9
  for b=1 to 9
    print a; "*"; b; "="; a*b
  next b
next a
end
```

程序流程图见图 2.24(a)和(b)所示。图中虚线框内是内层循环。

在这个程序执行过程中，a 的值为 1 时，b 从 1 变到 9，打印出 1*1=1，1*2=2，……，1*9=9；然后，a 变为 2，b 又从 1 变到 9，打印出 2*1=2，2*2=4，……，2*9=18；……。即 a 每变化一个值，b 要变化 9 次，外循环变量相对稳定，内循环变量变化比较快。在整个程序执行过程中，外循环体共执行 9 次，而内循环体共执行了 81 次。因此，与循环过程无关的操作，应尽量往外层次，以提高程序的效率。

例 2.6.8 求 $\sum_{I=1}^{20}\sum_{J=1}^{3}(I+1)^2(J+2)$。

采用双重 FOR 循环，可以直接写出如下程序(流程图见图 2.25)：

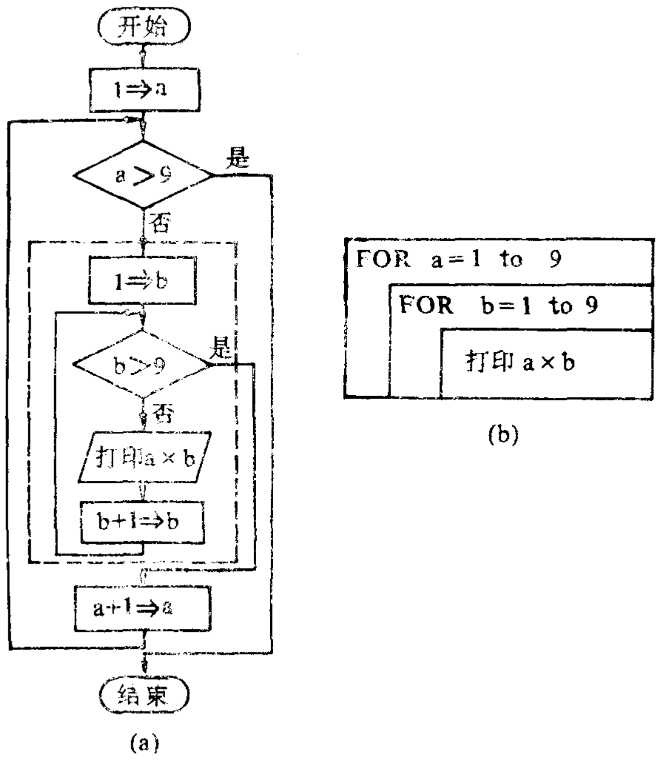


图 2.24

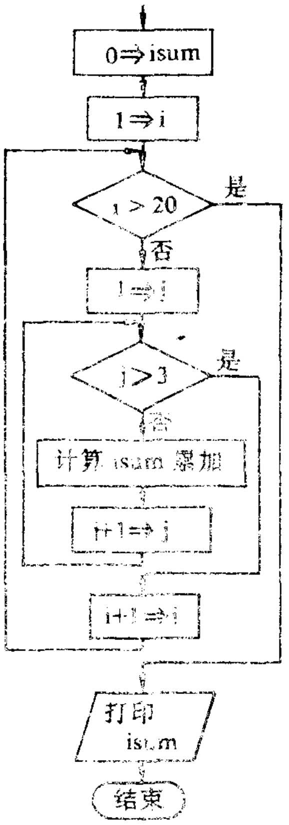


图 2.25

```
let isum=0
for i=1 to 20
  for j=1 to 3
    let isum=isum+(i+1)*(i+1)*(j+2)
  next j
next i
print "The sum is:"; isum
end
```

分析一下在执行循环时,一共要进行比较多少次?外循环变量由 1 变到 20 时,每一次在内循环中要进行 4 次"j>3?"的比较(循环变量 j 的值为 1,2,3,4),因此对"j>3"的判断检查共要进行 20×4=80 次,外循环的控制循环条件"i>20?",共要判断 21 次(外循环变量 i 的值为 1,2,3,…,20,21)。因此,整个程序包括 80+21=101 次判断比较。

如果把内外循环交换一下,程序改成下面这样,比较次数会有什么变化吗?

```
!
!
let isum=0
for j=1 to 3
   for i=1 to 20
      let isum=isum+(i+1)*(i+1)*(j+2)
   next i
```

```
next j
print "The sum is:"; isum
end
```

在这个程序中，执行外循环体 3 次，每一次要对"i>20?"进行比较判断 21 次，一共进行 3×21=63 次。再加上外循环对"j>3?"要进行比较判断 4 次，故总共进行比较判断 67 次。可以看到，两个算法的功能相同，但效率不同。一般说来，应当把变化范围大的循环变量放在内层。

在前面这个程序中，赋值语句中的(j+2)运算共进行了 60 次。若把(j+2)的运算搬到内循环之外(如下面所示)，则只要计算 3 次，减少了 57 次。

```
!
!
let isum=0
for j=1 to 3
    let k=j+2
    for i=1 to 20
        let isum=isum+(i+1) * (i+1) * k
    next i
next j
print "The sum is:"; isum
end
```

这就是前面提到过的，应尽量把与循环变量无关的运算移到循环外去。由于(j+2)与循环变量 i 无关，可以移到内循环外，而(i+1)·(i+1)的运算与 i 有关，不能移到内循环外。

如果把上述程序改成下面的程序，效率会进一步提高。

```
!
!
let isum=0
for j=1 to 3
    let k=j+2
      for i=1 to 20
        let m=i+1
        let isum=isum+m * m * k
      next i
next j
print "The sum is:"; isum
end
```

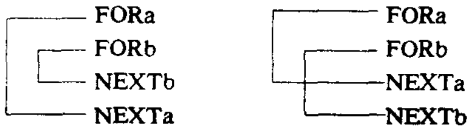
这时原来需要计算 120 次的(i+1)变为只需计算 60 次了。

在使用循环嵌套时，注意不可出现循环交叉，如图 2.26 所示。错误的嵌套往往由于把 FOR 与 NEXT 中的变量名没有按嵌套层次配对而造成的。

4. EXIT FOR 语句

EXIT FOR 语句在 FOR 循环中的作用，与 EXIT DO 语句在 DO 循环中的作用相同。

当执行 EXIT FOR 语句时，流程跳到 NEXT 语句后面一个语句处。



(a) 正确嵌套　　(b) 错误嵌套

图 2.26

例 2.6.9 五家共井：我国古代数学巨著《九章算术》卷第八方程第十三题为"今有五家共井，甲二绠(汲水用的井绳)不足，如(意为接上)乙一绠，乙三绠不足，如丙一绠；丙绠不足，如丁一绠；丁五绠不足，如戊一绠；戊六绠不足，如甲一绠。如各得所不足一绠。皆逮(dài，及，指达到水面)。问井深，绠长各几何。答曰：井深七丈二尺一寸。甲绠长二丈六尺五寸，乙绠长一丈九尺一寸，丙绠长一丈四尺八寸，丁绠长一丈二尺九寸，戊绠长七尺六寸。"

这是世界上最古老的不定方程问题。

设井深为 X，甲绳长为 A，乙绳长为 B，丙绳长为 C，丁绳长为 D，戊绳长为 E，可有如下方程组：

$$\begin{cases} 2A+B=X \\ 3B+C=X \\ 4C+D=X \\ 5D+E=X \\ 6E+A=X \end{cases}$$

化简上述方程组得：721C＝148X。

显然这个方程有无数组解。只有确定了井深 X，才能算出 C，乃至 A，B，D，E。但 X 也是未知的。然而，我国大多数地区水井深度的情形是：50 寸≤X≤1000 寸　并且已知所有绠长都是以寸为单位的整数。所以可以让 X 从 50 变到 1000，找出最先满足方程 721C＝148X 为整数的 C，进而算出 A，B，D，E。所以，程序一定是个重复结构。

```
!
!
for x=50 to 1000
  let c=148 * x/721
  if c=Int(c) then
    let d=x-4 * c
    let e=x-5 * d
    let a=x-6 * e
    let b=x-2 * a
    print "x="; x
    print"a="; a, "b="; b, "c="; c, "d="; d, "e="; e
    exit for
  end if
next x
end
```

Ok. run

```
x=721
a=265        b=191        c=148        d=129        e=76
Ok.
```

2.7 程序设计举例

可以说,程序设计语言就像供程序员转动的魔方。程序员可以按自己的意愿构造解题的蓝图。这一章介绍了 True BASIC 的基本控制结构和基本操作,有了这些基础,我们已经有可能在相当广泛的范围内进行程序设计了。下面介绍几种简单的算法设计方法。

2.7.1 穷举

在例 1.3.2 和例 2.6.5 中,讨论了求 1 到某个自然数范围内的素数的问题。在例 2.6.9 中,又讨论了五家共井问题。这两个问题的算法有一个共同之处,就是从某个数据集合中查找它的一个子集(或一个元素)。查找的办法是对该集合中的元素一个一个地进行检查。这种方法称为"穷举"。概括地说,穷举的基本思想是,对要解决的问题的所有可能情况一个不漏地进行检查,从中得到符合要求的答案。这是一种非常朴素的解题思想,可以用来解决"是否存在"和"有多少种可能"等类型的问题。穷举算法的结构多为循环。

例 2.7.1 从 10 名学生的成绩表中,找出不及格学生的姓名。

(1) 算法设计

首先可以把问题归结为如下算法:

S1:取 10 名学生的成绩表;

S2:找出不及格学生的姓名。

采用循环结构,可以把上述算法改为用伪代码表示如下:

```
FOR n=1 TO 10

取一名学生成绩 score

IF score<60 THEN print 该生姓名 name
NEXT n
```

下面确定如何输入学生成绩表。设学生成绩已预先知道,为了提高程序的执行速度,把学生成绩表数据预先存入程序中,用 DATA/READ 语句输入数据。

(2) 程序编码

根据上述算法及输入方式,可以得到如下程序:

```
! Find out no-passed student names
!
print "No-passed students are:"
for n=1 to 10
   read name $ , score
   if score<60 then print name $ ,
next n
```

```
!
data Zhang,95,Li,86,Wang,49,Zhao,65
data Chen,57,Guo,73,Lin,52,Ma,53
data Fang,89,Gau,100
end
```

```
Ok. run
No-passed students are:
Wang        Chen        Lin        Ma
Ok.
```

上述程序执行时,对 10 个同学的成绩一一进行检查,凡成绩不及格(<60)者,便打印出其姓名。

上述程序在穷举检查过程中,采用了计数循环,即 FOR-NEXT 循环。计数循环用于循环次数预知的情况。但是它的灵活性较低,如果学生数有所改变,就要修改程序。而且要一个一个地去数需要处理的数据个数,万一数错一个,便会出错。这时,可以充分利用逻辑函数。

例 2.7.2 从若干名学生的成绩表中,找出成绩最高的学生的姓名。

(1) 算法设计

可将问题抽象为

S1:从学生成绩表中,找出最高的成绩;

S2:打印成绩最高的学生的姓名。

首先对 S1 细化。这个子问题是,如何从若干个数据中找出一个最大值。这个问题有普遍性。在例 1.1.4 和例 1.2.4 中已经讨论了这类问题的算法,即输入一个数,与 max 比,如大于 max,则将它存入 max。根据这一思想,可以写出子问题 S1 的算法如下:

```
read name $ , score
let max=score
do
  read name $ , score
  if score>max then let max=score
loop until end data
```

用这个算法要对每一个学生的成绩都要检查一次,也是一种穷举法,所以使用了循环结构。

下面再对 S2 细化。在 S1 中,必须将全部学生的成绩都检查完才能得出 MAX 中放的成绩是否该成绩单中的最高成绩,然而成绩最高的同学又不一定排在成绩单的最后一位。所以找到了最高成绩后,还必须重新用穷举法检查哪位同学的成绩等于 MAX。当找到某一位同学的成绩等于 MAX 时,便结束检查,打印出这位同学的姓名。算法如下:

```
restore
do
  read name $ , score
```

```
  if score=max then exit do
loop until end data
print name $ ; "'s score is the best. "
```

(2) 程序编码。

根据上述给出的 S1,S2 再加上 DATA 语句,便可以得到下面的 True BASIC 程序:

```
! Find a student whose score is the best.
!
data Zhang,95,Li,86,Wang,49,Zhao,65
data Chen,57,Guo,73,Lin,92,Ma,53
data Fang,89,Gau,100
!
read name $ ,score
let max=score
do
  read name $ ,score
  if score>max then let max=score
loop until end data
restore
do
  read name $ ,score
  if score=max then exit do
loop until end data
print name $ ;"'s score is the best. "
end
```

```
Ok. run
Gau's score is the best.
Ok.
```

此题也可以只用一个单重循环求得成绩最高的学生的姓名,做法是在每次循环中,如果新读入的分数大于 max,则将此分数赋给 max,同时将 max 对应的学生姓名存入另一变量(如:name_of_max $)。这样,循环结束后,max 中存的就是最高分,而 name_of_max $ 中存的就是最高分学生的姓名。程序如下:

```
! Find a student whose score is the best.
data zhang,95,Li,86,Wang,49,Zhao,65
data Chen,57,Guo,73,Lin,92,Ma,53
data Fang,89,Gau,100
read name $ , score
let max=score
let name_of_max $ =name $
do
  read name $ ,score
```

```
  if score>max then
     let max=score
     let name_of_max $ =name $
  end if
loop until end data
print name $ ;"'s score is the best."
end
```

从上述两个问题中可以看出，在循环结构中使用 READ 语句是很经济的。这两个穷举问题都使用了单重循环。在实用中，还可以根据问题的性质，采用多重循环。

例 2.7.3 百钱买百鸡：公元五世纪末，我国古代数学家张丘建在《算经》中提出了“百鸡问题”：“鸡翁一，值钱五；鸡母一，值钱三；鸡雏三，值钱一。百钱买百鸡，问鸡翁、母、雏各几何？”意为“每只公鸡值 5 元，母鸡值 3 元，小鸡三只值 1 元；用 100 元买 100 只鸡，问公、母、小鸡各可买多少只？”

下面用逐步细化的方法来设计求解该题的程序。

(1) 确定总的解题策略，这个策略就是建立“百钱买百鸡”问题的数学模型。

“百钱买百鸡”可以用下面的方程表示：

$$cocks + hens + chicks = 100 \qquad (2.7.1)$$

$$5cocks + 3hens + chicks/3 = 100 \qquad (2.7.2)$$

其中：cocks——鸡翁数；hens——鸡母数；chicks——鸡雏数。

这是一个不定方程问题，未知数多于方程数。要解此不定方程应先固定一个变量的值，然后求其它两个值。即对剩下的两个变量的各可能值一一测定，看它是否满足方程(2.7.1)与(2.7.2)。显然这是一个穷举问题。那么应先确定哪一个变量呢？先看一下三个变量 cocks，hens，chicks 的可能值：

(i) cocks 的最小值可能为零，因为只要鸡母与鸡雏数满足百钱买百鸡条件这是允许的。若全部钱用来买鸡翁，只能买 20 只，但这是不符合百钱买百鸡条件，所以，鸡翁数必须小于 20，假设它为 19。故 cocks 的值为 0～19 中的整数。

(ii) hens 的值最小为零，即一百元只买鸡翁与鸡雏，hens 的值最大为 33，即全部买鸡母。

(iii) chicks 的值最小为零，最大为 100。

对 cocks，hens，chicks 三个未知数来说，cocks 取值范围最少。为提高程序的效率，先考虑对 cocks 的值进行穷举测试，得图 2.27(a)所示算法。

用循环算法对 cocks 的值进行穷举测试时，在每一次循环过程中，cocks 的值都是一定的。因此，下一步将要对“对应于一个 cocks 寻找满足百钱买百鸡条件的 hens 与 chicks”。这时，可以用下面的两种方法把解题过程细化：

第一种方法是解方程(2.7.1)、(2.7.2)组成的方程组得：

$$\begin{cases} hens = 25 - 7 * cocks \\ chicks = 75 + 75 * cocks \end{cases}$$

然后再检查 hens，chicks 是否为正整数。这种方法太麻烦，要人做的工作太多。

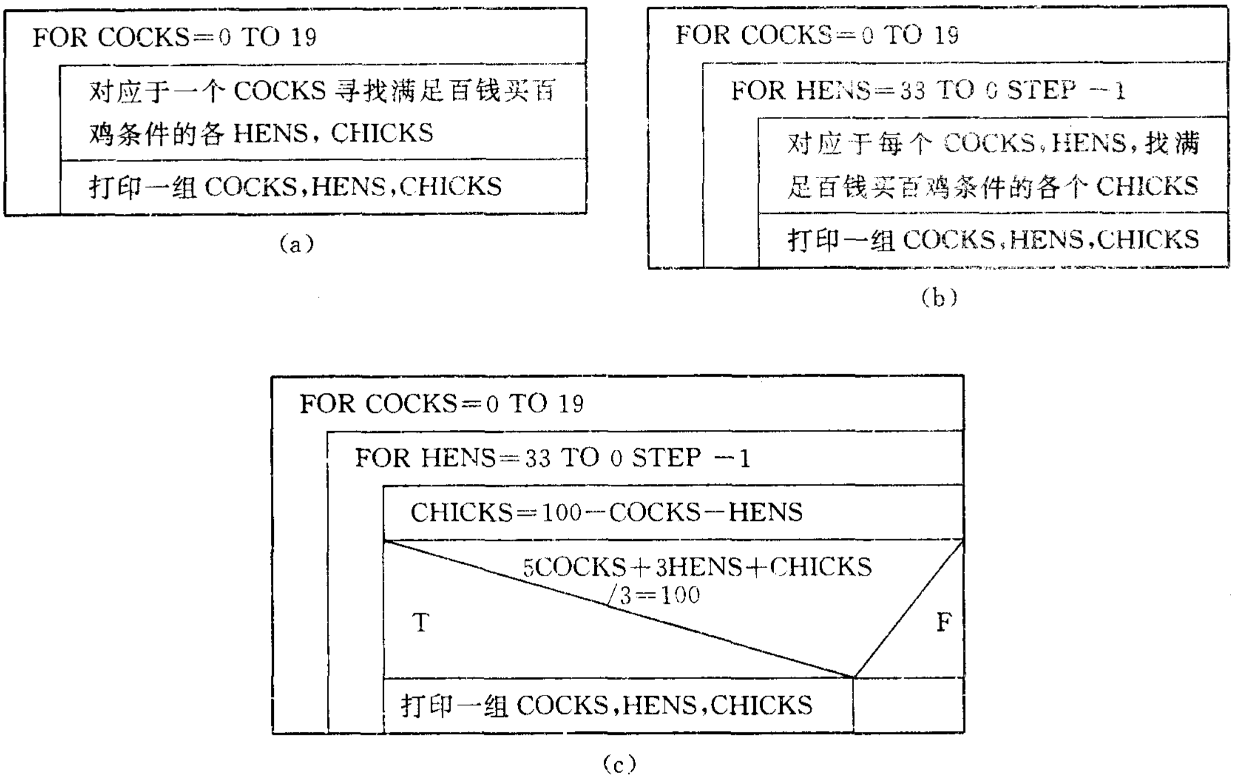


图 2.27

另一种方法是，在固定一个 cocks 的值的前提下，对 hens（因为 hens 的取值数少于 chicks）进行穷举测试，看对某一个 hens 值有哪些 chicks 的值能满足百钱买百鸡条件。这又是一层循环结构。经过这一步细化后的算法如图 2.27(b)所示。

下一步是解决这样一个问题：对于每个 cocks，hens，怎样去寻找满足百钱买百鸡条件的 chicks。由于 cocks，hens 值已设定，便可以由(2.7.1)式推得：

$$chicks=100-cocks-hens$$

这个 chicks 是相对于某一组 cocks，hens 的，即这时的 cocks，hens，chicks 是一组可能解，它只满足“百鸡”条件，还未满足“百钱”条件。是否真实解，还要看它们是否满足(2.7.2)式。于是算法又细化为图 2.27(c)。这时，全部算法都可以用 True BASIC 的基本操作代替了。

在本例中，由于要进行两层的穷举测试，所以算法结构是一个二重循环。

(2) 程序编码

根据图 2.27(c)所示算法，可以得到下面的 True BASIC 程序：

```
! Problem of hundred chickens.
!
print"cocks","hens","chicks"
for cocks=0 to 19
  for hens=33 to 0 step -1
    let chicks=100-cocks-hens
    if 5*cocks+3*hens+chicks/3=100 then
      print cocks,hens,chicks
```

```
        end if
        next hens
    next cocks
    end
```

```
Ok. run
cocks       hens        chicks
 0           25           75
 4           18           78
 8           11           81
 12          4            84
Ok.
```

2.7.2 迭代

所谓迭代(iterate)是指重复执行一组指令(或一定步骤,在每次执行这组指令时,都从变量的原值推出它的一个新值。

例如,求 $s=\sum_{n=1}^{100}n$,可以写出以下程序:

```
!
let s=0
let i=1
do
  let s=s+i
  let i=i+1
loop until i>100
print"s="; s
end
```

可以看出,s 的初值为零,每执行一次循环,在 s 的原值基础上增加一个 i 的值,因此 s 的值在不断变化。经过100次循环后,就得到最终的 s 值。这个方法就是迭代方法。就是说,解题的结果不是一次直接求出的,而是通过多次迭代,不断地由原值推出新值,直到达到要求为止。变量 s 称为迭代变量,它的值是不断变化的。同样,变量 i 也是迭代变量,它的值也是不断变化的,每次对 i 的原值加1得到 i 的新值。

迭代方法是计算机解题的一种基本方法。要善于将一个复杂的问题分解为一个迭代过程,用循环来处理迭代。计算机运算的特点是速度快,最适合作重复性的操作,它还有自动判断的功能,因此能控制其重复执行的次数。

在求代数方程、微分方程的数值解以及求函数值的插值等数学问题中迭代方法得到广泛的应用,这是一种数值近似求解方法。例如,在科学技术领域内,经常会碰到求解一元方程 $f(x)=0$的问题。这时许多人总是尽力去研究如何用解析表达式去精确地描述它的解。遗憾的是,能用解析法解出的一元方程实在是太有限了!即使 $f(x)$的形式比较简单,解析法也常常无能为力,象方程 $x-e^{-x}=0$以及 $f(x)$是5次以上的多项式时,都找不出用初等函数表示

的根的解析表达式。在这种情况下，人们可以采用一些近似解法：

(1) 隔离实根

方程 $f(x)=0$ 的根的分布可能是很复杂的。但是，如果函数 $f(x)$ 在 $[a,b]$ 上连续，且 $f(a)\cdot f(b)<0$（即在区间两端点函数值异号），那么至少有一个数 $\zeta\in[a,b]$，使得 $f(\zeta)=0$。若同时 $f(x)$ 的一阶导数 $f'(x)$ 在 $[a,b]$ 内存在且保持定号，则这样的 ζ 在 $[a,b]$ 内显然是唯一的。

根据这一定理，可以用逐步细分的方法（方法很多，如用逐次分半的方法），把一元方程式 $f(x)$ 的根分别隔离在不同的小区间内，如图2.28所示。把这个区间缩小到一定的程度，便可以得到一定精度的解。

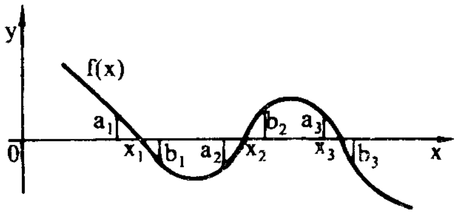


图 2.28 实根的隔离

(2) 用图解法求根的粗略近似值

把函数分隔成许多小区间以后，很容易作出函数在各个区间的图像来。从函数的图像上，能估计出方程的粗略近似值。

(3) 用迭代法使根精确化

迭代法的实质就是按照下列步骤构造一个序列 $x_0,x_1,\cdots,x_n,\cdots$，来逐步逼近方程 $f(x)=0$ 的解：

① 确定迭代格式，即建立迭代关系，需要将方程 $f(x)=0$ 改写为 $x=\varphi(x)$ 的等价形式；

② 选取适当的初值 x_0；

③ 构造序列 $x_0,x_1,\cdots,x_n,\cdots$，即先求得 $x_1=\varphi(x_0)$，再求 $x_2=\varphi(x_1)$，…，如此反复迭代，就得到一个数列：

$$x_{k+1}=\varphi(x_k)$$

即 $x_0,x_1,x_2,\cdots,x_n,\cdots$。若这个数列收敛，即存在极值，且函数 $\varphi(x)$ 连续，则很容易得到 $x^*=\lim\limits_{k\to\infty}x_k$。这个极限值 x^* 就是方程 $f(x)=0$ 的根。

例2.7.4 用二分法求方程 $x^2-x-2=0$ 在区间(0,3)内的一个根。

(1) 算法设计

二分法是用计算机求解多项式方程时，常用的一种方法。其基本思想如图2.29所示。若 $f(x_1)$ 与 $f(x_2)$ 的符号相反，则方程 $f(x)=0$ 在区间 (x_1,x_2) 至少有一个根。取 x_1 与 x_2 的中点 $r_1=\frac{1}{2}(x_1+x_2)$，将区间分为两半。然后比较 $f(r_1)$ 与 $f(x_1)$ 的符号，若符号相同，则根必在 (r_1,x_2) 之间，否则在 (x_1,r_1) 之间。这样每作一次二分法，含根区间恰好缩小一半。不断重复二分过程无限多次时，含根区间将缩为一点。显然这是不可能的。但是有一点是可以肯定的，即每重复二分一次，所得的中点 ROOT 与根的距离便会越近。不断重复二分，便会构造出一系列的中点 ROOT1，ROOT2，…，ROOTn，…，当某个中点 ROOTn 与根的距离小于规定的误差 ε_0 时，该 ROOTn 就是所解方程的近似解。根据上述分析，可以得到图2.30(a)所示的算法。

这里还有两个问题需要进一步细化：

① 如何求误差(ERROR)；

② 如何判定 $f(x_1)$ 与 f(ROOT)是否同号。

首先讨论如何求 ERROR。ERROR 是 ROOT 的精确解之间的差。遗憾的是，精确解是

不知道的。如果知道了精确解，也就不须求 ROOT 了。因此要寻找其它计算 ERROR 的方法。

由图2.29可知，$|x_1-ROOT|$大于 ROOT 与精确解之间的差值。当$|x_1-ROOT|\leqslant\varepsilon_0$时，ROOT 与精确根之间的差也一定小于 ε_0。所以可以用$|x_1-ROOT|$来代替 ROOT 与精确根之间的差值。

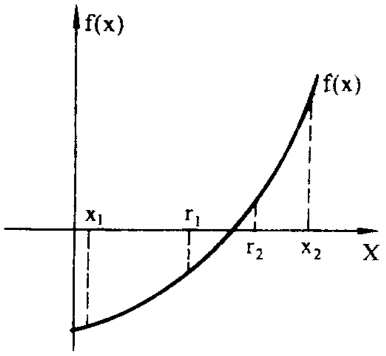


图 2.29

下面再讨论如何判定 $f(x_1)$与 f(ROOT)是否同号。我们知道，若 $f(x_1)$与 f(ROOT)同号，则应有

$$f(x_1)*f(ROOT)>=0$$

或有

$$sgn(f(x_1))=sgn(f(ROOT))$$

那么，如何计算 $f(x_1)$与 f(ROOT)呢？由于题目中已经给定了 f(x)的形式，所以很容易写出：

$$f(x_1)=x_1*x_1-x_1-2$$

$$f(ROOT)=ROOT*ROOT-ROOT-2$$

根据上述讨论，可以把图2.30(a)细化为图2.30(b)。

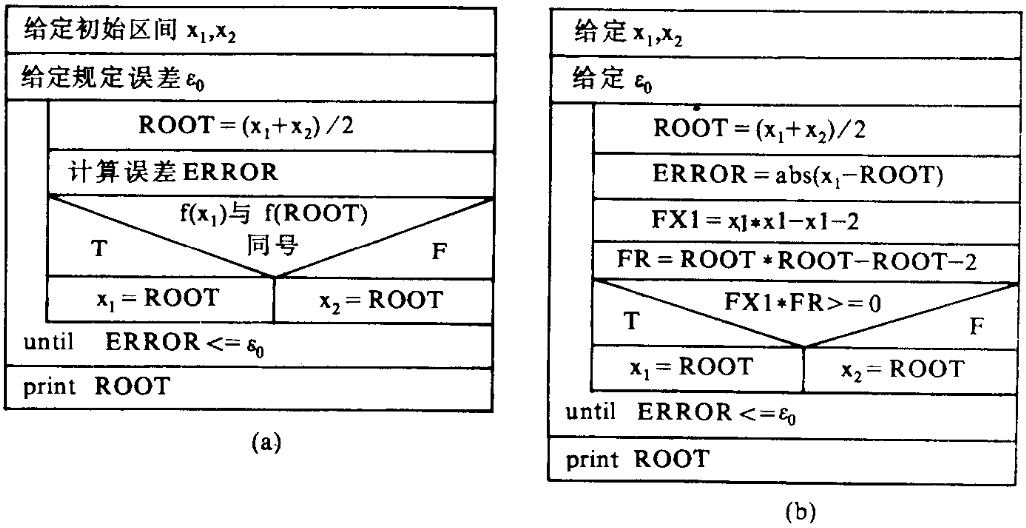


图 2.30

(2) 程序编码

由图2.30(b)可以得出如下 True BASIC 程序：今假定给定的 $\varepsilon_0=\frac{1}{2}\times10^{-5}$。

```
! To find the root of a equation in a given
! interval by interval halving method.
!
let x1=0
let x2=3
let eo=0.5e-5
do
```

```
  let root=(x1+x2)/2
  let error=abs(x1-root)
  let fx1=x1 * x1-x1-2
  let fr=root * root-root-2
  if fx1 * fr>=0 then let x1=root else let x2=root
loop until error<=eo
print"Root between 0 & 3 is:"; root
end
```

```
Ok. run
Root between 0 & 3 is: 2.
Ok.
```

例2.7.5 用牛顿迭代法求一元方程 $x-e^{-x}=0$的根。

(1) 算法设计

牛顿迭代法又称牛顿切线法。它比一般迭代法具有更高的收敛速度，从而成为更加有效的方法。

设 $f(x)=0$的曲线如图2.31所示，它与 x 轴的交点为 x^*。

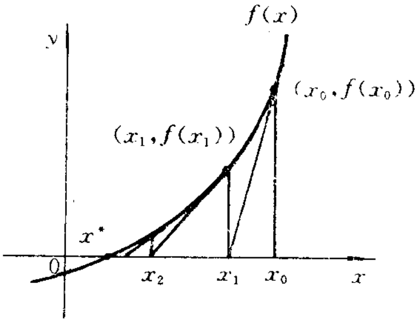


图 2.31 牛顿迭代法的几何意义

首先用根隔离法找到 x^* 存在的区间，如果函数 $f(x)$在此区间内连续可导，则可以用牛顿迭代法来求 $f(x)=0$的近似根 x^*。方法是将根所在区间的一个边界点赋给变量 x_0，过点$(x_0,f(x_0))$作函数 $f(x)$曲线的切线与 x 轴交于 x_1，则此切线的斜率为 $f'(x_0)=f(x_0)/(x_0-x_1)$，即

$$x_1=x_0-\frac{f(x_0)}{f'(x_0)},$$

显然，所得到的 x_1更接近于 x^*，继续过点$(x_1,f(x_1))$作 $f(x)$的曲线的切线，与 x 轴相交于 x_2，则 $x_2=x_1-\frac{f(x_1)}{f'(x_1)}$，重复上述过程，可得迭代公式：

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \qquad (2.7.3)$$

当给定根的初始值 x_0和允许误差 ε 后，就可以根据以上迭代公式得到一个符合误差要求的近似实根。

牛顿迭代法的算法结构如图2.32(a)所示。

在图2.32(a)中还需要进一步确定 $f(x_0)$与 $f'(x_0)$。在题目中已给定：

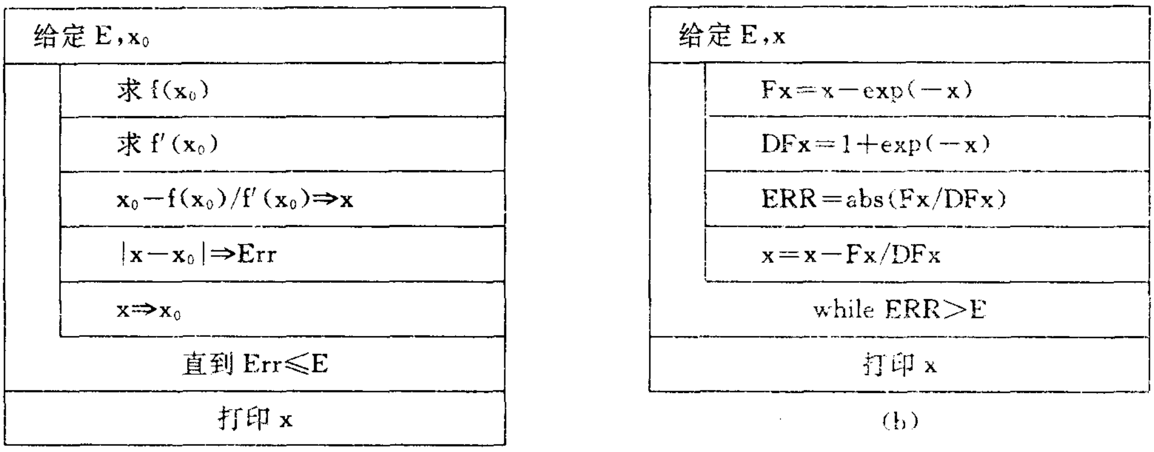
$$f(x)=x-e^{-x}$$

用 True BASIC 表示 $f(x_0)$为

$$Fx = x0 - \exp(-x0) \qquad (2.7.4)$$

$f'(x_0)$为

$$DFx = 1 + \exp(-x) \qquad (2.7.5)$$



(a) (b)

图 2.32

由于已知 $x=x_0-f(x_0)/f'(x_0)$，因此 $x-x_0=-f(x_0)/f'(x_0)$，所以求误差 $|x-x_0|$ 可以用 $|-f(x_0)/f'(x_0)|$ 代替。即 ERR＝|Fx/DFx|。

按照式(2.7.4)及(2.7.5)并考虑到迭代是用新值代替旧值的操作，对图2.32(a)细化可得图2.32(b)所示算法。

(2) 程序编码

对图2.32(b)所示算法细化，并用 True BASIC 语言描述，可得如下程序：

```
! To find root of a equation by Newton recurrence method.
!
print "Fractional error allowend E:";
input e
PRINT "first approximate X0:";
input x
do
  let fx=x-exp(-x)
  let dfx=1+exp(-x)
  let err=abs(fx/dfx)
  let x=x-fx/dfx
loop while err>e
print"x=";x
end
```

```
Ok. run
Fractional error allowend E: ? .01
first approximate X0: ? 1
x=.567143
Ok.
```

通过前面的两个例子，可以看出，在迭代算法中，至少存在一个不断直接或间接地由旧值递推出新值的变量，这个变量就是“迭代变量”。上面程序中的 fx,dfx,err,x 都是迭代变

量。构造迭代算法的关键就是确定迭代变量并建立迭代关系式。

可以用递推的方法来建立迭代关系。即找到如何从变量的前一个值推出其下一个值的公式(或关系)。例如，前面求 $\sum_{n=1}^{100} n$ 的程序中，"let s=s+i"和"let i=i+1"就是递推公式。

应当说明，递推与迭代是有联系而有区别的两个概念。递推不一定采取迭代，例如，由 a 的值推出 b 的值，由 b 的值推出 c 的值，……，这是递推，但是 a,b,c 是不同的变量，不存在用一个新值去代替变量的原值。因此，这不是迭代，但递推常常采用迭代方法处理，以便用计算机循环处理。

下面举例说明如何用递推方法来建立迭代关系。例：例1.3.3中讨论过的求多项式：

$$s=\frac{1}{1!}+\frac{1}{2!}+\frac{1}{3!}+\cdots+\frac{1}{(i-1)!}+\frac{1}{i!}+\frac{1}{(i+1)!}+\cdots+\frac{1}{100!}$$

可用递推形式表示计算过程。设 s_i 为累加到第 i 项时 s 的值，例如 $s_1=1$，$s_2=1+\frac{1}{2}$，$s_3=1+\frac{1}{2}+\frac{1}{6}$，…，第 i 项的值以 t_i 表示，例如：$t_1=1$，$t_2=\frac{1}{2!}$，$t_3=\frac{1}{3!}$，有以下关系：

$$\begin{cases} t_1=1 & s_i=t_i \quad (i=1) \\ t_i=\dfrac{t_{i-1}}{i}, & s_i=s_{i-1}+t_i \quad (i>1) \end{cases}$$

写成 true BASIC 程序段如下：

```
let t=1
let s=t
for i=2 to 100
  let t=t/i
  let s=s+t
next i
```

这就建立了迭代关系。

我们知道，利用泰勒公式或其它方法，可以得到某些初等函数的幂级数展开式，用来对这些初等函数进行近似计算，如

$$\sin x=x-\frac{x^3}{3!}+\frac{x^3}{5!}-\frac{x^7}{7!}+\cdots+(-1)^n\frac{x^{2n+1}}{(2n+1)!}+\cdots$$

$$\cos x=1-\frac{x^2}{2!}+\frac{x^4}{4!}-\frac{x^6}{6!}+\cdots+(-1)^n\frac{x^{2n}}{(2n)!}+\cdots$$

$$e^x=1+x+\frac{x^2}{2!}+\cdots+\frac{x^n}{n!}+\cdots$$

等。把这些近似公式变换为递推形式并不困难，这样就可以用迭代法进行计算了。

例2.7.6 用牛顿迭代公式计算 $\sqrt{a}$，计算精度为10^{-5}(即误差不超过10^{-5})。其中 a=1，2,3,…,10。

(1) 算法设计

首先把1～10中各数的平方根的计算过程抽象为图2.33(a)所示算法。下面再考虑如何求 $\sqrt{a}$。

题中要求使用牛顿迭代公式。用牛顿迭代公式求 $\sqrt{a}$，需要先写出求解方程。由于 x=

$\sqrt{a}$，即 $x^2=a$，可得

$$f(x)=x^2-a=0$$

然后用牛顿迭代法解此方程，求根 x。

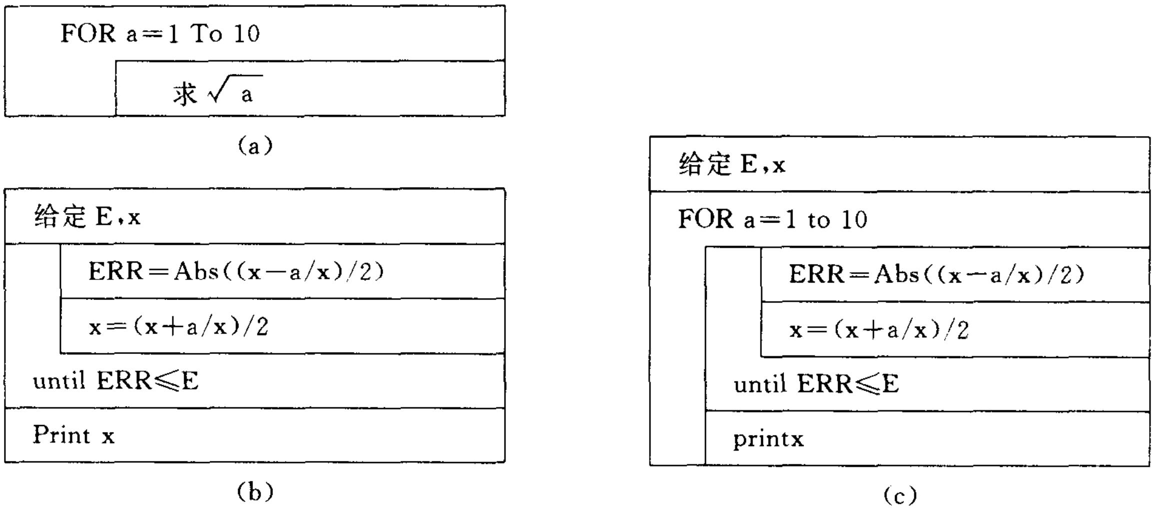


图 2.33

已知牛顿迭代公式为

$$x_{n+1}=x_n-\frac{f(x_n)}{f'(x_n)}$$

令 $$f(x_n)=x_n^2-a,f'(x_n)=2x_n$$

故 $$x_{n+1}=x_n-\frac{x_n^2-a}{2x_n}=\frac{1}{2}\left(x_n+\frac{a}{x_n}\right)$$

这就得到了求 $\sqrt{a}$ 的牛顿迭代公式。用它对图2.33(a)中的"求 $\sqrt{a}$"细化便可得到图2.33(b)所示用牛顿迭代法求 $\sqrt{a}$ 的算法。其中误差的求法与例2.7.5相同，即 $ERR=|f(x_n)/f'(x_n)|=|(x_n^2-a)/2x_n|=\left|\frac{1}{2}\left(x_n-\frac{a}{x_n}\right)\right|$，图中表示为 ERR=ABS((x-a/x)/2)。

将图2.33(b)代入图2.33(a)中，便得到本题的算法，如图2.33(c)所示。图中把"给定 E，x_0"提到了 FOR 循环之前。这是因为题意要求所求各数的平方根的绝对误差是要求相同的，且假定对每次求 $\sqrt{a}$ 时给出的方程 f(x)的粗略根(第一次给出的近似根 x_0)也相同。即"给定 E，x_0"与循环过程无关。把与循环过程无关的操作提到循环之前，可以提高程序执行的速度。

(2) 编码

根据图2.24(c)，用 True BASIC 描述的程序及运行结果如下：

```
! This program calculates the square roots
! of numbers from 1 to 10 using the Newton's
! recurrence formula
!
input prompt"Fractional error allowend E:": e
input prompt"First approximate X0:": x
for a=1 to 10
```

```
  do
    let err=Abs((x-a/x)/2)
    let x=(x+a/x)/2
  loop while err>e
  print"The square root of"; a; "is:"; x
next a
end
```

```
Ok.run
Fractional error allowend E: .001↙
First approximate X0: 3↙
The square root of 1 is: 1.
The square root of 2 is: 1.41421
The square root of 3 is: 1.73205
The square root of 4 is: 2.
The square root of 5 is: 2.23607
The square root of 6 is: 2.44949
The square root of 7 is: 2.64575
The square root of 8 is: 2.82843
The square root of 9 is: 3.
The square root of 10 is: 3.16228
Ok.
```

例2.7.7 用矩形法求定积分 $\int_0^{\pi}\sin x dx$

(1) 算法设计

定积分 $\int_a^b f(x)dx$ 的几何意义就是 f(x)和 f(a),f(b)及 x 轴所围成的面积。矩形法就是将此面积分割成若干小区间的面积(如图2.34),在每个小区间内用矩形面积来近似函数 f(x) 在此区间上的定积分,并将所有小矩形的面积累加做为 $\int_a^b f(x)dx$ 的近似值。分割的区间越小,则计算结果越精确。

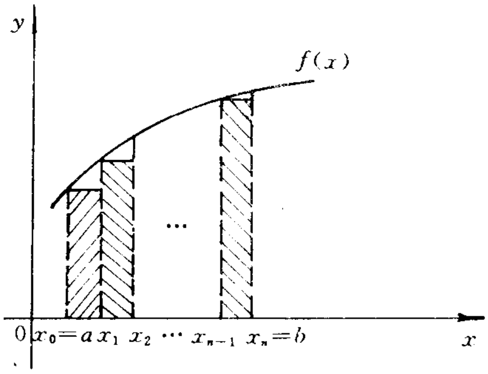


图 2.34

现将区间分成长度相等的 n 个小区间,则每个小区间的长度为

$$h=(b-a)/h$$

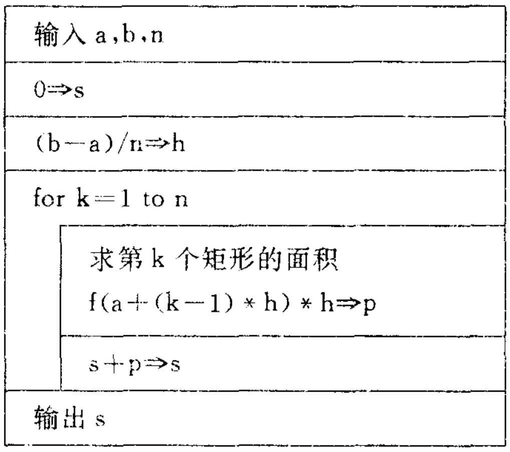
第 k 个小矩形的面积为

$$f(x_{k-1})*h=f(a+(k-1)*h)*h$$

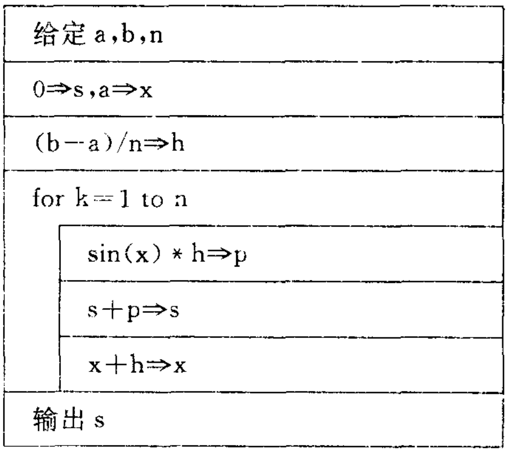
用 s_k 表示前 k 个矩形面积之和,可建立如下的迭代公式:

$$s_{k+1}=s_k+f(a+k*h)*h \quad (2.7.6)$$

算法结构如图2.35(a)所示。



(a)



(b)

图 2.35

对图2.35(a)细化并考虑到远算效率,即得到求$\int_0^\pi \sin x dx$ 的算法(见图2.35(b))。

(2) 程序编码

```
! to caculate definite integral
! rectangle method
!
do
  input prompt "n=":n
loop until n>1
let a,s=0
let b=pi
let x=a
let h=(b-a)/n
for k=1 to n
  let p=sin(x) * h
  let s=s+p
  let x=x+h
next k
print "s=";s
end
```

```
Ok. run
n=10
s=1.98352
Ok. run
n=100
s=1.99984
Ok. run
n=1000
s=2.
```

用计算机求定积分也可采用梯形法。梯形法与矩形法的不同之处在于：对于每一个小区间，梯形法是用梯形面积来近似函数在此区间上的定积分（如图2.36）。同样将区间分成 n 等分，则第 k 个小梯形的面积为

$T_k=(f(x_{k-1})+f(x_k))*h/2$，其中 $x_k=a+k*h$

用 S_k 表示前 k 个梯形面积之和，则迭代公式为：

$$S_{k+1}=S_k+T_{k+1} \qquad (2.7.7)$$

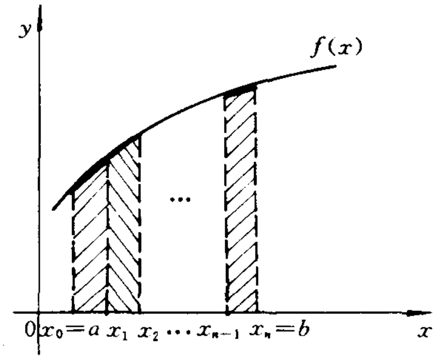


图 2.36

梯形法的收敛速度比矩形法快，因此也更实用。关于梯形法的算法及程序在此不再详述，有兴趣的读者，可考虑自己完成。

按照算法本身提供的精确性，可以把迭代分为精确迭代与近似迭代两类。有相当多的问题，对其求精确解的算法找不到或难于实现，如例2.7.4、例2.7.5、例2.7.6等，而不得不从某个已知值，顺着可以逼近其精确解的路径，用迭代法求出其精确解的近似值。这类迭代就称为近似迭代。近似迭代法也称逐次逼近法。一般说来，迭代无限多次后，便可以求出解的精确值。但是由于运算有时间性限制，而且工程技术上也允许有一定的误差，实际的迭代过程总是有限的。只要迭代过程收敛，总可以估计出近似解的误差范围。当这个误差范围小于给定的值时，便可以结束迭代过程了。

精确迭代是指算法本身就能提供问题的精确解。一般说来，在构造迭代的同时已经可以算出所需迭代次数了，并且常常用它来控制迭代过程。这类问题的运算，若结果有误差，也不是算法本身造成的，而是计算机用有限的字长表示实数而造成的。

例2.7.8 小猴吃桃问题的迭代解法。

在例1.3.4中，曾给出了小猴吃桃问题的递推式：

$$\begin{cases} PEACHS_{n-1}=(PEACHS_n+1)\times 2 & (1\leqslant n<10) \\ PEACHS_{10}=1 \end{cases}$$

以 PEACHS 为递推变量。上面的递推表达式建立了一个迭代关系。可用下面的迭代程序求解。在迭代求解以前，要先给出 PEACHS 的初始值，即 peachs＝1。由于它是第10天的桃子数，要求出第1天的桃子数要迭代10次，所以执行10次循环。

```
let peachs=1
for day=9 to 1 step-1
  let peachs=(peachs+1) * 2
next day
print"The first number of peachs is:"; peachs
end
```

```
Ok. run
The first number of peachs is: 1534
Ok.
```

例2.7.9 求阶乘。阶乘的定义是

$$\begin{cases} FACT_n=1 & (n=0,1) \\ FACT_n=n!=n(n-1)\cdots 2\cdot 1 & (n>1) \end{cases}$$

也可以写成下面的递推形式：

$$\begin{cases} FACT_n=1 & (n=0,1) \\ FACT_n=FACT_{n-1}\cdot n & (n>1) \end{cases}$$

根据上述递推表达式很容易得到图2.37所示循环算法。该算法中 FACT=FACT * I 所需要的循环次数，是由输入的 N 决定的。该算法很简单，由它不难写出 True BASIC 程序。请读者自己完成。

输　入　N	
FACT=1	
FOR I=1 TO N	
	FACT=FACT * I
打　印　FACT	

图　2.37

2.7.3　确定性模拟

数字模拟又称数字仿真，是指在实验室条件下，利用电子计算机系统模拟真实的物理、化学、生物环境以及其它各种事物或事物的变化。数字模拟能够解决数学分析难以解决或不易解决的问题。当直接研究真实系统费用太高、时间太长、实现太难、危险太大甚至完全不可能时，采用电子计算机进行数字模拟，再现真实系统及其变化，并在速度上将其加速或减慢，在空间上将其放大或缩小，常常能获得比较满意的效果。

数字模拟的过程是：

(1) 提出问题，分析问题；

(2) 构造模拟模型；

(3) 设计并实现模拟算法；

(4) 分析模拟结果。

建立模拟模型是进行数字模拟的关键。按照模拟模型的类型，数字模拟可分为确定性模拟与概率性模拟两类。这一小节先介绍确定性模拟。

确定性模拟所使用的确定性模型，通常以数学方程的形式出现。它所描述的系统多是连续系统。这些方程往往难以求解或无法求解，但却可以在计算机上用模拟的方法获得近似解。进行模拟时，要先用时间步长法或事件步长法对其进行离散化处理，然后用数值方法求解。

例2.7.10　导弹追击飞机

导弹追击飞机时，弹头在任一瞬间均要指向飞机。由于飞机在飞行，导弹必须不断变换飞行方向，才能保持弹头指向飞机。这是一个确定性问题。但是它的数学方程的求解却是很困难的。用计算机对飞机和导弹的运动进行模拟，可以得到这个问题的近似解。

为简化问题，假定飞机作水平飞行，导弹也在同一平面内飞行。如图2.38所示。当飞机出

现在坐标原点(0,0)时，导弹从(X_0,Y_0)处开始追踪攻击飞机。

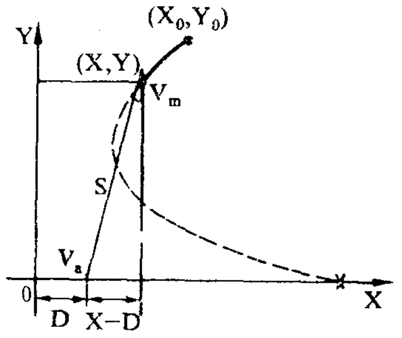


图 2.38

由于导弹与飞机飞行的轨迹都是时间的函数，所以采用时间步长法进行模拟。设

DT——时间步长；

(X,Y)——任一瞬间导弹的坐标；

V_n——飞机的飞行速度；

V_m——导弹的飞行速度；

(D,0)——任意瞬间飞机的坐标；

S——任意瞬间导弹与飞机间的距离。

由图2.38所示导弹与飞机间的位置关系，可以导出

$$S=\sqrt{Y^2+(X-D)^2}$$

导弹飞行方向角 θ 有

$$\sin\theta=(X-D)/S$$

$$\cos\theta=Y/S$$

每经过一个时间步长 DT，导弹与飞机的位置将发生如下变化：

$$X \Leftarrow X-V_m * DT * (X-D)/S$$

$$Y \Leftarrow Y-V_m * DT * Y/S$$

$$D \Leftarrow D+V_a * DT$$

这就是按时间步长得到的导弹追击飞机的模拟模型。用迭代算法，很容易得出当导弹与飞机间的距离 S 小于某一预定距离 S_0(近似认为已经追上飞机)时，导弹追踪的时间及导弹与飞机之间的实际距离。

根据上述分析，不难写出模拟程序如下：

```
! A program of simulation:
!     A missile makes an attack to an aircraft
!
print "x0, y0, va, vm, dt, s0";
input x0, y0, va, vm, dt, s0";
let x=x0
let y=y0
let d=0
let t=0
let s=sqr(y * y+(x-d) * (x-d))
do while s>=s0
   let d=d+va * dt
   let x=x-vm * dt * (x-d)/s
   let y=y-vm * dt * y/s
   let s=sqr(y * y+(x-d) * (x-d))
   let t=t+dt
loop
```

```
print "T="; t, "S="; s
END
```

```
Ok. run
X0, Y0, Va, Vm, Dt, S0? 40000, 100000, 500, 1000, 1, 500↙
T=112            S=438.778
Ok. run
X0, Y0, Va, Vm, Dt, S0? 40000, 100000, 500, 1000, 1, 50↙
T=188            S=49.4746
Ok. run
X0, Y0, Va, Vm, Dt, S0? 40000, 100000, 500, 10000, 1, 5↙
T=781            S=.290676
Ok. run
X0, Y0, Va, Vm, Dt, S0? 40000, 100000, 500, 10000,.001,5↙
T=10.597         S=1.22075
Ok.
```

由程序的运行结果可以看出：S_0的选择应以时间步长 DT 的选择为前题：当 DT 较大而S_0太小时，模拟系统会产生振荡，误差上升；同一 S_0下，选择得 DT 越小，模拟越准确。

这个模拟程序稍加修改，便可以打印出导弹飞行的轨迹。此外，这种模拟还可以扩展到三维空间以及飞机作机动飞行的情形。

2.7.4 概率性模拟

1. 随机变量与随机数：概率性模拟用于以概率模型描述的系统。这种系统是一种随机系统。其特征是系统所处的状态是在离散时刻发生变化的，例如十字路口的交通流量，公共汽车站的等车人数，电话交换机的电话呼叫，原子反应堆里中子穿透铅壁的运动等。

任何一个随机过程中至少有一个量是随机变化的，这个量称为随机变量。在一个随机过程中，随机变量的值是一个随机数序列。概率模拟，就是用计算机产生的随机数序列模拟随机事件中的随机数序列。但是，计算机与实际随机过程的物理机制是不可能相同的，难以产生出真随机数。因而，常常要用计算机产生一些“伪随机数”来代表实际情况。伪随机数看起来好象是随机的，人们并不知道计算机给出的一个伪随机数究竟是什么数。但是，实际上，计算机系统提供的随机数序列是用一个复杂的公式递推出来的。这个随机数序列是可以重复产生的。即下一次使用计算机时，自然可以得到一个相同的随机序列。使用伪随机数序列的好处在于它是可重复的，因而允许进行重复试验。而真随机数的序列一般是不重复出现的。后面提到的随机数便是指伪随机数而言。

最常用的随机数是均匀分布的随机数，它们相当均匀地分布在最大值与最小值之间，并“随机”地出现。所谓均匀分布，就是随机变量 X 所出现的值，在它变化的区间中的一切值上都具有出现相等的概率。例如，由计算机产生的一个伪随机数序列，包含1000个随机数，它们的值在0～1之间，则在由0～0.5和0.5～1两个区间内各约有随机数500个。

True BASIC 中有一个产生随机数的程序。在 True BASIC 程序中用函数 Rnd 调用“随机数产生程序”，可以得到均匀分布在[0,1)间的随机数(即$0\leqslant Rnd<1$)。需要再次指出的

是，Rnd 只能给出伪随机数。

例2.7.11

```
! A example of pseudo-random number sequance
1
for i=1 to 5
   print Rnd
next i
end
```

```
Ok. run
.548508
7.82692e-2
.687876
.335258
.951922
Ok. run
.548508
7.82692e-2
.687876
.335258
.951922
Ok.
```

可以看到，每执行一次程序，True BASIC 都给出同一个随机数序列中的前5个数，两次运行结果完全相同。

如果在使用 Rnd 之前执行一条 RANDOMIZE 语句，情况就不同了，True BASIC 每执行一次程序，都将给出不同的伪随机数序列。RANDOMIZE 称为随机化语句，它的作用是使下一次得到的伪随机数序列不同于上一次的伪随机数序列。

例2.7.12

```
! A program with statemetn RANDOMIZE
randomize
for i=1 to 5
      print Rnd
next i
end
```

```
Ok. run(第一次执行)
.40071
.697051
.372866
.746937
.787595
```

```
Ok. run(第二次执行，产生与第一次不同的随机数序列)
.444662
.246474
.96912
.320072
.350494
Ok.
```

Rnd 只能给出[0,1)间的随机数。为了得到任何区间的随机数，可用以下方法：

(1) 用 Rnd * n 可以得到[0,n)之间的随机数。

(2) 想得到一个0到9之间的随机整数，可用 Int(Rnd * 10)，因为 Rnd 的值为0～1之间的一个数。乘以10，再用 Int 函数取整，就能得到区间[0,9]之间一个整数。想得到一个0～n之间的随机整数，可用 Int(Rnd * (n+1))。

(3) 如想得到一个[101,200]之间的随机整数，可用 Int(Rnd * 100)+101因为 Int(Rnd * 100)得到一个0～99之间的整数，再加101，得到101～200之间的一个随机数。

如果想得到一个[n,m]之间的随机整数，可以用 Int(Rnd * (m−n+1))+n。

例2.7.13 抽取样本。

某次质量检查时，要从成品库中的 n 台产品中抽取 m 台送检。抽样时可以将这 n 台产品编号为1～n，然后用下面的抽样程序确定送检产品的号码。

```
!  Program for sampling.
!
randomize
print "Enter the number of products";
input n
print "Enter the number of sampling";
input m
print "Sampled numbers is:"
for i=1 to m
   print Int(Rnd * N+1)
next i
end
```

```
Ok. run
Enter the number of products? 100↙
Enter the number of sampling? 5↙
Sampled numbers is:
82
2
14
71
69
Ok.
```

2. 概率性模拟：概率性模拟的实质就是用随机数来模拟系统过程的随机性。概率性模拟一般采用三种形式：(1)时间步长法；(2)事件步长法；(3)蒙特卡洛法。

分述如下：

(1) 概率模拟的时间步长法：在模拟过程中，以时间段为单位研究过程的变化状况，称为时间步长法。这一方法在例2.7.10中已经使用过了，不过那是一种确定性模拟问题。在概率模拟中，时间步长法就是模拟每经过一个时间步长，判断是否有指定的事件发生。利用时间步长法进行概率模拟的例子很多，银行存款情况、商店顾客买物情况、车辆交通流、电话交换台与用户呼叫匹配、各种排队问题等。下面用一个简单的例子来说明时间步长法在概率模拟中的应用。

例2.7.14 加油站汽车到达情况模拟：一个汽车加油站内设置的油泵的台数，决定于汽车到达加油站的情况。因此，为了决定加油站内应设置几台油泵，应先了解汽车到达加油站的情况。加油站的汽车总是随机到达的。下面来模拟汽车到达加油站的情形。

假定这个加油站的来车率是每小时20辆，那么当选定时间步长 TIME_STEP(单位为分)后，汽车到达加油站的模型可表达为

1个时间步长内来一辆车的概率＝20/60 * TIME_STEP

1个时间步长内不来车的概率＝1－20/60 * TIME_STEP

注意，时间步长选择得太长，可能使得模拟结果不合理。如若设时间步长为1小时，每隔1小时才判断一次有没有汽车到来，就不大合理，它无法区分出是在第59分钟同时来20辆车呢？还是每3分钟均匀地来一辆车。一般说来，时间步长选得足够小才能得到合理的模拟结果。但是时间步长选得太小会增加计算的工作量，模拟过程花费的时间过长。因此时间步长的选择应视具体问题并在精度和计算时间之间折衷而定。有时也可以选几个不同时间步长并经过比较确定。

时间步长确定之后，便可以用一个随机数序列模拟随机系统的一个时间序列中每个时间步长中的状态。即对应于每个时间步长，从序列中取出下一个随机数，来代表该时间段中系统的状态。本题假设 TIME-STEP≤3分钟，即平均有一辆车到达的时间间隔，这样20/60 * TIME_STEP，是一个≤1的数，计算机产生的随机数是一个[0,1)范围内的数，如果随机数小于20/60 * TIEM_STEP，就表示有一辆车到达(为方便起见，假定汽车刚好在时间步长的末尾到达)。若随机数大于或等于20/60 * TIME_STEP 就表示该时间步长内没有汽车到达。

根据上述分析可以得到如下算法：

给定时间步长 TIME_STEP

```
DO
  产生一个随机数 RND
  IF RND<20/60 * TIEM_STEP THEN
      有车
  ELSE
      无车
  打印一次结果
UNTIL 不再模拟
```

对上述算法细化。

① 有车与无车：用一个变量 AUTOMOBILE_NUM 来给汽车按到达先后顺序排号。某时间步长内有车到达，则编号加1，否则车号不变。

② 打印一次结果：结果要求给出每个时间步长内的结果。设开始模拟的时间为零(TIME＝0)，则在模拟过程中每经一个时间步长，模拟时间有：

$$\text{TIEM}=\text{TIME}+\text{TIME_STEP}$$

对应于每个时间应打出相应的到达的车号。

③ 停止模拟：停止模拟的条件可以很多，如到车数达到某个值或超过某个时间长度等。这里采用后者，并要求用户自己定义模拟的时间长度 TIEM-LONG。

根据上述讨论，可得到下面的细化了的算法。

给定时间步长：TIME_STEP；

给定模拟时间长度：TIEM_LONG；

```
DO
  产生一个随机数 RND
  IF RND<20/60 * TIEM_STEP THEN
    LET AUTOMOBILE_NUM=AUTOMOBILE_NUM+1
  END IF
  LET TIME=TIME+TIME_STEP
  PRINT TIME, AUTOMOBIL_NUM
LOOP UNTIL TIME>TIME_LONG
END
```

对上述算法再细化：

① 增加 RANDOMIZE 语句；

② 给各变量赋初值；

③ 增加注释和中间计算语句；并代之以 True BASIC 语句，可得如下程序：

```
! simulation: automobiles arrive at a gas station.
!
print"How long to simulation";
input time_long
print "Enter the time step";
input time_step
randomize
let t, automobile_num=0
let arr=20/60 * time_step
print "time", "automobile_num"
do
  let r=Rnd
  if r<arr then
      let automobile_num=automobile_num+1
  end if
```

```
16   let t=t+TIME_STEP
17   print t,atuomobile_num
18 loop until t>=time_long
19 end
```

```
Ok. run
How long to simulation? 30↙
Enter the time step? 2↙
time          automobile_num
2              1
4              2
6              3
8              4
10             5
12             5
14             6
16             7
18             7
20             7
22             8
24             9
26             9
28             9
30             10
Ok.
```

上述程序只适用于时间步长小于3分钟的情形。若选时间步长大于3分钟，请读者自己考虑。

(2) 概率性模拟的事件步长法：事件步长法是以事件步长为增量，来模拟系统的动态过程。其原理基本与时间步长法类似。

例2.7.15 中子扩散问题。

原子反应堆的壁是铅制的。中子从壁内侧（为简单，设以垂直方向）进入，走一定距离（设为铅原子直径 D）后，与铅原子碰撞。之后改变方向（这个方向是随机的），又走一定距离（仍设为常数 D）后，与另一个铅原子碰撞。如图2.39所示，中子每次与铅原子碰撞后都以一个随机角 $\theta(0,2\pi)$ 弹射。经多次与铅原子碰撞，可能返回反应堆内，也可能穿出铅壁造成辐射，还可能被铅壁吸收，其结果是随机的。显然铅壁设计得越厚，穿透的概率就越小，被铅壁吸收的概率就越大。为了经济而安全地设计原子能反应堆的铅壁，需要导出铅壁厚与中子穿透率之间的关系来。但是这一问题用解析的方法是难以解决的。下面用计算机对中子的穿壁运动进行模拟。

为了建立概率模拟模型，首先分析一个中子在铅壁内的运动情况。

中子在壁内的运动与其每次与铅原子碰撞后的弹射角 θ 有关，由图2.39可知这个角是随机的，即

$\theta = 2 * PI * RND$

设中子在壁内与某一铅原子碰撞时距内壁的距离为 X，则下一次碰撞前 X 产生的变化为

$$X \leftarrow X + D * COS(2 * PI * RND) \quad (2.7.7)$$

设反应堆的壁厚为 M * D，每个中子在铅壁内碰撞 TIMEO 次后其能量就会被堆铅原子吸收，那么当碰撞次数 TIMES≥TIMEO 时，就可由 X 的变化决定它是被吸收了(0≤X≤M * D)，还是返回到反应堆(X<0)，或是扩散到了反应堆外(X>M * D)如表2.12所示。

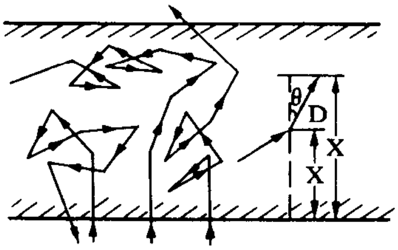


图 2.39　中子穿壁运动模型

表 2.12

0≤X≤M * D	T	F	F	TIMES≥TIMEO	T	F	F
X<0	F	T	F	穿　　透			X
X>M * D	F	F	T	吸　　收	X		
TIMES<TIMEO	F	T	T	返　　回		X	

采用事件步长法对中子的穿壁运动进行模拟时，可以选相当多的中子数(Num-NEUTR)，以对每个中子穿壁运动的模拟为一个事件步长。每给一个事件步长后记下累计的穿透数 NP、吸收数 NA 和返回数 NR，再进入下一个事件步长。

当按预定数目对中子穿壁运动全模拟完后，便可计算出该反应堆中中子的穿透率 NP%、吸收率 NA%和返回率 NR%。算法如图2.40(a)所示。

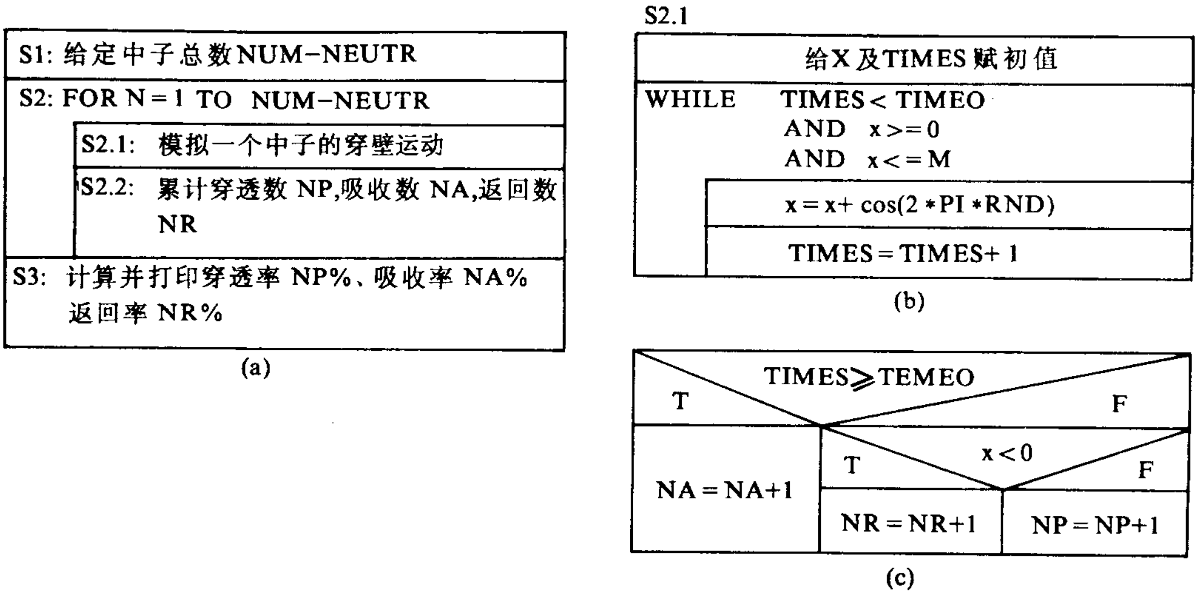


图 2.40

下面首先对 S2.1细化。S2.1是模拟一个中子的穿壁运动。在此模拟过程中，又可以把每

碰撞一次作为一个事件步长。每模拟一次碰撞，累计一次穿透距离 X 与碰撞次数 TIMES，直到可得出该中子穿透(X>M * D)、返回(X<0)或吸收(0≤X≤M * D，且 TIMES≥TIMEO)为止。对一个中子的穿壁过程的模拟算法如图2.40(b)所示。

下面再对 S2.2求精。S2.2是累计穿透数 NP、吸收数 NA 和返回数 NR。但是由 S2.1出来有三种情形，到底是哪种情形，要在累计之前先按表2.12判断。算法如图2.40(c)所示。

最后还要考虑如何计算并打印 NP%，NA%与 NR%。这一点是很容易的，因为只要知道了 NP，NA，NR 及总中子数 NUM-NEUTR，便可以计算出

$$NP\% = NP/NUM_NEUTR$$

$$NA\% = NA/NUM_NEUTR$$

$$NR\% = NR/NUM_NEUTR$$

设 X 的单位为铅原子的直径 D，TIME＝10。根据上述算法描述，再细化，便可以得到如下 True BASIC 程序及运行结果：

```
! Program of simulation:
! Problem of neutrons'diffusion in a reactor.
!
randomize
print "Input total of neutrons";
input num_neutr
print "Input deep of the reactor wall";
input m
print "Na%", "Nr%", "Np%"
for n=1 to num_neutr
  let x=0
  let times=0
  do while times<10 and x>=0 and x<=m
    let x=x+cos(2 * pi * Rnd)
    let times=times+1
  loop
  if times>=10 then
    let na=na+1
  else if x<0 then
    let nr=nr+1
  else
    let np=np+1
  end if
next n
let na=na/num_neutr * 100
let nr=nr/num_neutr * 100
let np=np/num_neutr * 100
print na, nr, np
end
```

```
Ok.run
Input total of neutrons? 1000↙
Input deep of the reactor wall? 10↙
Na%              Nr%              Np%
  17.2             82.8             0
Ok.run
Input total of neutrons? 1000↙
Input deep of the reactor wall? 5↙
Na%              Nr%              Np%
  18.6             80.4             1.
Ok.run
Input total of neutrons? 1000↙
Input deep of the reactor wall? 2↙
Na%              Nr%              Np%
  3.4              77.7             18.9
Ok.
```

显然，铅壁越厚，吸收率越高，穿透率越低。

(3) 蒙特卡洛法(Monte_Carlo Method)：蒙特卡洛法是一种模拟随机事件发生的技术。它几乎与任何领域都有关联，并涉及到概率在问题求解中的应用。蒙特卡洛法往往涉及到大量运算，显然不使用计算机想用它进行问题的求解是不可思议的。

例2.7.16 求 π 的近似值：求曲线 f(x)在[a,b]间的面积(如图2.41所示)，通常用定积分

$$S=\int_a^b f(x)dx$$

蒙特卡洛法则是一个统计实验方法。其步骤为

① 先用一个长为(b－a)，高为 c 的矩形，把 S 围在内；

② 由计算机产生 N 个均匀分布在矩形框内的随机点(x,y)，即 x 在[a,b]间均匀分布，y 在[0,c]间均匀分布；

③ 统计落入欲求面积内的随机点的个数 M；

④ 由公式 S＝M/N * c * (b－a)计算所求面积的近似值。c * (b－a)为矩形面积，$\frac{M}{N}$为落在欲求面积的范围内的概率，$\frac{M}{N}$ * c * (b－a)就是近似求出的面积。N 愈大，计算愈精确。

下面介绍如何用此法求 π 值。已知圆面积公式为

$$S=\pi R^2$$

当 R＝1时，S＝π。所以只要求出 $x^2+y^2=1$所包围的面积，就能求出 π 的值。为简便，先求出 $\frac{1}{4}$ 单位圆的面积(见图2.42)然后乘以4。算法如图2.43(a)所示。其中 S2，S3可以细化为图2.43(b)。

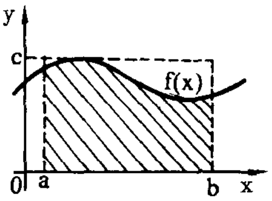


图 2.41

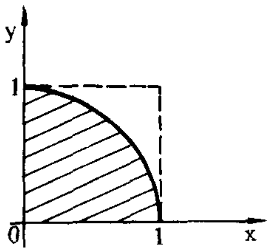


图 2.42

S1：给定随机点数 N
S2：产生各随机点
S3：统计阴影范围内的点数
S4：计算 π 值，打印

(a)

FOR I=1 TO N	
	产生随机数 X
	产生随机数 Y
	判断点(x,y)是否在阴影范围内并统计

(b)

图 2.43

```
! Compute the P1 with the Monte-Carlo method
!
randomize
print "How many spots";
input n
let m=0
for i=1 to n
  let x=Rnd
  let y=Rnd
  if x*x+y*y<=1 then let m=m+1
next i
print "Pi="; m/n*4
end
```

```
Ok. run
How many spots? 10↙
PI=3.6
Ok. run
How many spots? 10000↙
PI=3.1152
Ok. run
How many spots? 50000↙
PI=3.12856
Ok.
```

显然，所取点的密度愈高，计算结果愈准确。

习　题

2.1.1 试述 True BASIC 程序的组成。

2.1.2 "注释是程序中无关紧要的部分",这句话对吗?为什么?

2.1.3 什么叫窗口?True BASIC 能提供几种窗口?它们各有何用途?

2.1.4 解释执行方法与编译执行方法各有何特点?True BASIC 如何把它们统一起来?

2.2.1 什么叫数据?数据类型有什么意义?True BASIC 能提供哪几种数据类型?

2.2.2 什么叫数据结构?为什么进行程序设计时还要确定数据结构?

2.2.3 计算机程序中变量的意义是什么?

2.2.4 试述 True BASIC 命名变量名的规则。下列哪些是合法的 True BASIC 变量名,哪些是非法的?

A+B,5A3,SIN,INT,XπT,+5,AB$C5,ELSE,DATA,DTE$,TIME,SEX。

2.3.1 指出并改正下列 True BASIC 表达式中的错误。

(1) Y+2/X+4　　$\left(\text{表示}\dfrac{Y+2}{X+4}\right)$

(2) XY^3+1×1　　(表示 $(X\cdot Y)^3+1\times1$)

(3) P(P−A)(P−B)　　(表示 P(P−A)(P−B))

(4) AM−E^X SIN (2π+A)+2COS(25°+B)

(表示 $Am-e^x\cdot\sin(2\pi+\alpha)+2\cos(25°+\beta)$)

2.3.2 指出下列表达式的值:

(1) Abs(Int(−13.2)+Round(.32 * 5))

(2) Remainder(13,5)+Mod(13,5)

(3) Sqr(Truncate(163.256,−1))

(4) Sqr(Round(94.08765,2))

(5) "CAT">"DOG"

(6) "x(2)"<"x2"

2.3.3 将下列数学表达式写成 True BASIC 表达式。

(1) $ab+2x^2+(2^X)^3$;

(2) $\dfrac{V}{R_1+\dfrac{R_2\cdot R_3}{R_2+R_3}}$;

(3) $\dfrac{12x+5y-\sqrt{D^3-\left(\dfrac{m}{2}\right)^2}}{e^{-X}\cdot\text{arc tg}5}$;

(4) $2^m\sin^2x+\cos^2x$;

(5) $2\pi\sin(\pi+\alpha)+3\cos30°$

(6) $\text{Ln}(x+\sqrt{1+x^2})$

2.3.4 用一个 True BASIC 表达式表示下列各题。

(1) 产生一个两位随机奇数(即11~99之间的奇数)。

(2) 以 x(x>0)值为真数,以2为底的对数值。

(3) 将任意一个两位数 N 的个位数与十位数对换。如 N=62,则表达式的值应为26。

(4) 将变量 A 的值按四舍五入保留小数点后两位。

2.4.1 写一个程序,能在屏幕上显示出两个写在屏幕范围内的"欢迎"二字

提示:问一问老师,你所使用的显示器屏幕上可显示多少行、多少列字符。

2.4.2 设计一个打印信封的通用程序。

(1) 打印英文信封；

(2) 打印中文信封。

2.4.3 在下面的程序中填写正确的语句。

程序要求显示做52+49的算术竖式及横式。即显示：

```
   52
+  49
------
  101
52+49=101
```

程序如下：

```
let A=52
let B=46
let C=____
print ____
print ____
print "____"
print c
print ____
end
```

2.4.4 指出下面语句中的错误

(1) let x=5: y=6: z=7

(2) let r+2=v-3: x+y+z=r * v

(3) let a $ =flog

(4) let b=name $

(5) let c=1357

(6) input "a,b,c="; a; b; c

(7) read a $; b; c; d $; e $; f; g $
　　data wan; zhang; 35; 1988
　　1988-35; 2/3

(8) A $ =HAPPY
　　print A

2.4.5 在DATA语句中写了表达式或用表达式回答INPUT语句的提问时，会出现什么错误？

2.4.6 指出下面程序的运行结果：

(1)
```
read a, b, c
data 1, 2, 3, 4
let a=a+b+c
restore
read b, c, d, e
restore
data 5, 6, 7, 8, 9, 10
read e, f, g, h, i, j, k, l, m
print a, b, c, d, e, f, g, h, i, j, k, l, m
end
```

(2)
```
let a=1
let b=2
let c=3
let a=b
let b=c
let c=a
print a, b, c
end
```

(3)
```
let r=5
let r=r+2
print r
let r=r * 2
```

```
let r=r/2
print r
end
```

(4)
```
input x, y
let x=x+y
let a=x
let y=x+y
let x=y-x
let y=y-x
let x=y-a
let y=y/a
print x, y
end
```

(5)
```
let x$="GO ␣"
let y$="TO ␣"
let z$=x$&y$&"SCHOOL"
let a$, b$, c$=z$
let a$[7:12]="HOME"
let b$[1:0]="LET'S ␣"
let c$[7:6]="YOUR ␣"
print z$, a$
print b$, c$
end
```

2.4.7 甲、乙、丙三人上街买糖果。三人都买好后，甲对乙、丙说，我可以按你们现有的糖果数再送你们每人一份。甲送给乙、丙后，乙也按甲、丙现有的糖果数，送给甲、丙每人各一份糖果。丙也如此送了甲、乙各一份。互相赠送后。每人恰好各有64颗糖果。问甲、乙、丙原来各买了多少糖果？请编写 True BASIC 程序回答。

2.5.1 已知 prov1$="shanghai", prov2$="shanxi", prov3$="sichuan", a=6, b=12。请给出下列逻辑表达式的值：

(1) prov1$>prov2$ and prov3$<prov1$

(2) prov1$<prov3$ and not prov2$<prov3$

(3) 2*a<=b

(4) a<=b and b<=6

(5) not (a>b)

(6) a<=6 or b<=6

(7) a>12 OR(a<12 and b>6)

(8) b<5 and not a>13

2.5.2 把下面的命题用逻辑表达式表示：

(1) A 小于 B 或小于 C；

(2) A 和 B 都大于 C；

(3) A 和 B 中有一个小于 C；

(4) A 是非正数；

(5) A 是奇数；

(6) B 不能被 A 整除；

(7) 角 A 在第一或第三象限；

(8) A 是一个带小数的正数，而 B 是一个带小数的负数。

2.5.3 指出下列等式是否成立？为什么？

(1) not(A and B or not A and not B)=A and not B or not A and B

(2) A and B or not A and not B=(A or not B) and (not A or B)

(3) 已知 A and B=A and C, A or B=A or C，问 B=C 是否成立？

2.5.4 给出下列命题的逻辑表达式

(1) A,B,C 是三个二进制数码，写出 A=B=C 的条件。

（2）某宿舍住着 A,B,C,D 四个人。请根据哪些人在宿舍，写出宿舍里没有人唱歌的条件。这四个人的情形是：

① A 从来不唱歌；

② B 和 D 只有 A 在场时才唱歌；

③ C 在任何情况下，甚至一个人时也唱歌。

2.5.5 指出下列语句中的错误，并改正

```
(1) if r≥.5 then x=x+1 else x=x-1
(2) if r≤.5
      then x=x+1
      else x=x-1
    end
(3) if a=b$ then
      print A, B$ , C$
    else
      A=5:B$ =ADA:C$ =A+B$
    if end
(4) Select case x=a * b+c * d
      case x=2 or 3
           print "a"
      case x=4 and 5
           print "b"
      case x$ ="xyz", "rst"
           print"type"
      else
           print "end"
    end case
(5) let A=5
    select case X
        case is>A
          print 2 * A
        case is=A
          print A
        case is<A
          print A/2
    end select
```

2.5.6 由键盘输入三条线段的长度，求这三条线段所构成的三角形的面积。

2.5.7 由键盘输入三个任意数，按从大到小的顺序把他们输出。

2.5.8 按下述规定编一计税程序：

（1）货价≤2000元免税；

（2）2000元<货价≤5000元，抽税2%；

（3）货价>5000元的，除抽税2%以外，超过5000元的部分再抽税5%，外加手续费60元。

2.5.9 从键盘输入一个年份号，判断它是否闰年。

2.5.10 下表所示的判定表为某销售公司对顾客订货单的处理原则。

购货金额	任意	≥500	<500	任意	<500	≥500
过去交款情况	好	不好	不好	好	不好	不好
库存物资是否够	够	够	够	不够	不够	不够
立即发货	√		√			
要求先付款		√				
暂挂,到货后发货				√	√	
退回						√

请设计一个 True BASIC 程序,当输入某顾客订货单金额、过去交款情况及库存物资状况后,能立即给出处理方案。

2.5.11 某航空公司规定:在旅游旺季7~9月份,如果订票超过20张,优惠票价的15%;20张以下,优惠5%;在旅游淡季1~5月份、10月、11月份,订票超过20张,优惠30%,20张以下,优惠20%。

设计一个 True BASIC 程序能根据月份和旅客订票张数决定优惠率。

2.6.1 指出下列程序段中的错误

(1)
```
do while a * a<a+a
   a=a+1
loop
```

(2)
```
for I=1 to 5
   read A, B, C
   for J=1 to 3
      data 3 * PI
   next I
next J
```

(3)
```
for I=J+1 TO 2 * J STEP 0.1 * J
   print I,J
   let J=-J
next I
```

2.6.2 若 a,b 分别具有初值3和100,试指出下面的程序段执行结束后,a,b 的值各为多少?程序的打印结果是多少?

(1)
```
do while a<=b
   let a=a * a
loop
print a
```

(2)
```
do while a * b< >500
   let b=b+25
loop
print b
```

(3) let b=77

```
let a=3
Do
  print a
  let a=a+2
loop while a<=sqr(b)
end
```

(4) for k=1 to 5

```
  print "k="; k
  let b=b+1
  read a
  print "a="; a
    if b=7 then exit for
  data 1, 2, 3, 4, 5, 6, 7, 8, 9
next k
print"b="; b
end
```

(5) let a=1

```
let b=1
for k=2 to 4
    for J=1 to 1
        let b=b * J
    next J
    let a=a+b
next k
print"a="; a
end
```

(6) let a=1

```
let b=2
let s=0
for I=10 to 6
    let s=s+a
    let c=a+b
    let a=b
    let b=c
next I
print s
end
```

(7) for I=b to a step -10

```
    let b=a+b
    let a=b+a
    let x$=""
    for I=b to a step -1
```

```
        let x$ =x$ +"x"
        print x$ ;
    next I
    print
next I
(8) for I=a to b step 10
    let b=a+b
    let a=b+a
    let x$ =" "
    for I=a to b
        let x$ =x$ +"x"
    next I
    print x$
next I
```

2.6.3 按下列要求修改例2.6.7给出的程序。

(1) 打印一个矩形的九九乘法表。

(2) 打印一个直角三角形的九九乘法表。

2.6.4 有一个数列，它的头三个数为0，0，1，以后的每个数都是其前三个数的和。请编程序打印此数列，直到第30个数或最后一个数超过1010为止。

2.6.5 相传古代印度国王要褒奖他的聪明能干的宰相达依尔（国际象棋发明者），问他要什么？达依尔回答："陛下只要在国际象棋棋盘的第一个格子上放一粒麦子，第二个格子放二粒麦子，以后每个格子的麦子数都按前一格的两倍计算。如果陛下按此法给我64格的麦子，就感激不尽，其它什么也不要了。"国王想："这还不容易！"让人扛了一袋麦子，但很快用光了，再扛出一袋还不够，请你为国王算一下共要给达依尔多少小麦？（设1立方米小麦约1.4×10^8颗）

2.6.6 各单位到银行取工资时，为了分发方便，要领回各种面值的钞票。请设计一个根据某单位每人实领款数统计各种面值钞票数的程序。

2.6.7 编一个可以输出任何精度的程序，输出：(1) 两个数的商；(2) 两个数的积。

提示：求商时，模仿手算，从最高位开始，逐次得到商的各位数字。求积时，将两个数从个位起，每三位划一段，以乘数的每一段去乘被乘数中的每一段。在计算过程中，应注意每一次相乘所得的部分积在整个积中的位置，以及每一次相乘时有无进位。

2.6.8 两种不同的药片，一种重量11mg，另一种重10mg。这两种药片外形相同，且装在相似的药瓶中，由于工作人员疏忽，将若干瓶尚未贴上药签的药瓶混放在一起了。现在要求用天平只称一次，把两种药瓶完全区分开。

提示：用二进制判别法，即

① 先把所有药瓶编号；

② 从第i瓶中取出2^{i-1}片药片；

③ 称出所有取出药片的总重量；

④ 以某药片为标准，算出所取药片的标准重量；

⑤ 将标准重量与实际重量之差化为二进制数；

⑥ 根据二进制规律判断区分两种药。

2.6.9 编一个程序，计算 N 个自然数的最大公约数。

2.6.10 True BASIC 中语句的格式通常为〈语句关键字〉〈语句体〉。试编一个程序模拟 True BASIC 检查语法错误(可以先编对某特定语句(如 let 语句)的错误检查的模拟程序，并且可以先规定变量名只用两个字符)。

2.7.1 从所有学生成绩中打印出成绩最低的学生的姓名。

2.7.2 编一个能对任意整数进行因数分解的程序。

2.7.3 各因数之和等于其本身的正整数，称为完备数，编一个求1000之内的所有完备数的程序。

2.7.4 把一元钱全兑换成硬币，有多少种兑换方法?

2.7.5 百马百瓦问题：有100匹马驮100块瓦，大马驮3块，小马驮2块，两个马驹驮1块。问有大马、小马、马驹各多少?

2.7.6 搬砖：三十六块砖，三十六人搬，男搬四、女搬三、两个小儿抬一砖，要求一次全搬完，问须男、女、小儿各若干?

2.7.7 奇妙的算式：下面是两道奇妙的算式，相同的字母或汉字代表相同的数字，不同的字母或汉字代表不同的数字。请写出它们代表的实际算式。

(1)
```
   EGAL
 ×    L
 ------
   LAGE
```

(2)
```
      习学好好
 ×      们我
 -----------
    上向天天们我
```

2.7.8 谁去执行紧急任务：侦察班长接到一项紧急任务，准备在代号为 A,B,C,D,E,F 六名战士中挑选若干名人员去执行该任务。人员配备的要求是：

(1) A,B 两人中至少去一人；

(2) A,D 不能一起去；

(3) A,E,F 三人中要派两人去；

(4) B,C 两人都去或都不去；

(5) C,D 两人中去一人；

(6) 若 D 不去，则 E 也不去。

2.7.9 国王分财产：一个国王临终前给儿子们分财产。他把财产分为若干份，然后给第一个儿子一份，再加上剩余财产的$\frac{1}{10}$；给第二个儿子两份，再加上剩余财产的$\frac{1}{10}$；……；给第 i 个儿子 i 份，再加上剩余财产的$\frac{1}{10}$。每个儿子都窃窃自喜。以为得到了父王的偏爱，孰不知国王是“一碗水端平”的。请用程序回答，老国王共有几个儿子，共分给儿子们多少财产。

2.7.10 验证哥德巴赫猜想：1742年6月7日，德国数学家哥德巴赫(C. Goldbaeh 1690～1764)在给彼得堡的大数学家欧拉的信中问道：“任何大于6的偶数均可表为两个整素数的和吗?”，同年6月30日欧拉复信道：“任何大于6的偶数都是两个奇素数之和，这一猜想，我还不能证明，但我确信无疑地认为这是完全正确的定理”这就是至今尚未被证明的哥德巴赫猜想。请编程序验证哥德马赫猜想。

2.7.11 1982年我国有10亿3千万人口。以年人口平均增长率分别为2%，1.5%，0.5%计算，从1982年起经过几年我国人口就会增长到12亿。

2.7.12 计算圆周率的级数方法有

(1) 1605年英国数学家瓦里斯把 π 表示成

$$\frac{\pi}{2}=\frac{2}{1}\cdot\frac{2}{3}\cdot\frac{4}{3}\cdot\frac{4}{5}\cdot\frac{6}{5}\cdot\frac{6}{7}\cdot\frac{8}{7}\cdot\frac{8}{9}\cdots\cdot\frac{2n}{2n-1}\cdot\frac{2n}{2n+1}\cdots$$

(2) 17世纪德国数学家莱布尼兹把 π 表示成

$$\frac{\pi}{4}=1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\cdots+(-1)^{n-1}\frac{1}{2n-1}+\cdots$$

用来求 π 的级数除上面两个之外，尚有

(3) $\pi=2\cdot\dfrac{2}{\sqrt{2}}\cdot\dfrac{2}{\sqrt{2+\sqrt{2}}}\cdot\dfrac{2}{\sqrt{2+\sqrt{2+\sqrt{2}}}}\cdots$

(4) $\dfrac{\pi^2}{8}=1+\dfrac{1}{3^2}+\dfrac{1}{5^2}+\dfrac{1}{7^2}+\cdots+\dfrac{1}{(2n-1)^2}+\cdots$

(5) $\dfrac{\pi}{6}=\dfrac{1}{2}+\dfrac{1}{2}\cdot\dfrac{1}{3\cdot 2^3}+\dfrac{1\cdot 3}{2\cdot 4}\cdot\dfrac{1}{5\cdot 2^5}+\dfrac{1\cdot 3\cdot 5}{2\cdot 4\cdot 6}\cdot\dfrac{1}{7\cdot 2^7}+\cdots$

(6) $\dfrac{\pi^2}{6}=1+\dfrac{1}{2^2}+\dfrac{1}{3^2}+\dfrac{1}{4^2}+\cdots+\dfrac{1}{n^2}+\cdots$

(7) $\dfrac{\pi}{2}=1+\dfrac{1}{3}+\dfrac{1}{3}\cdot\dfrac{2}{5}+\dfrac{1}{3}\cdot\dfrac{2}{5}\cdot\dfrac{3}{7}+\dfrac{1}{3}\cdot\dfrac{2}{5}\cdot\dfrac{3}{7}\cdot\dfrac{4}{9}+\cdots$

(8) $\dfrac{\pi}{2}=\left(1+\dfrac{1}{1\cdot 3}\right)\left(1+\dfrac{1}{3\cdot 5}\right)\left(1+\dfrac{1}{5\cdot 7}\right)\cdots$

试根据上述任一个无穷级数，设计一个计算 π 的近似值的 True BASIC 程序，要求误差小于10^{-5}。你能说明上述哪一种级数的收敛速度较快吗？

2.7.13 计算圆周率的正多边形方法有

(1) 随着圆内接正 n 边形的边数 n 的增加，它的面积逼近圆的面积。

(2) 随着圆外切正 n 边形的边数 n 的增加，它的面积也逼近圆的面积。

试用上述一种方法求 π 的近似值，要求误差小于10^{-5}。

2.7.14 让计算机充当一位算术老师，给学生们出一系列的两位数的加法题，当学生的答案是正确的时候，计算机就在终端上回答“对了，做得好！”，如果学生做错了，计算机就回答“错了。另做一题”。

2.7.15 将上题改为能给出四则运算题目的数学机器。

2.7.16 弹簧的上端固定，垂直悬挂1kg 的重物，平衡后弹簧伸长2cm。假定用力将重物从平衡位置上向下拉 x cm 时，作用在它上面的弹力为4x。试用0.01秒为步长，模拟将弹簧再拉长1cm 时，重物的运动。

2.7.17 在例2.7.7中，假设飞机也带有导弹，该导弹能以800米/秒的速度飞行，并带有对付来袭导弹的装置，它的方向朝着第一枚导弹。假定飞机在40秒后施放自带导弹，试模拟飞机和两枚导弹的运动。

2.7.18 模拟旅程；一位持月票上班者每天早晨的旅程大致如下：

走到地铁车站：	用3分钟
等车：	平均用0～5分钟
坐车：	平均　7～10分钟
走出地铁车站：	4分钟

请用 True BASIC 程序模拟此人某天的旅程及总共花费的时间。

2.7.19 计算机模拟摸球：在口袋里放有手感相同的3只红球、4只白球，随机地摸出3只球来，记录这三只球的色别，然后放回口袋中，共摸500次，问摸到3只都是红球或都是白球的机率是多少？

2.7.20 生日问题：有人说，一个50人的班级，至少有两个人生日相同（月份不要求相同）的概率为97%，即100个50人的班级中，有97个班级有同生日的学生的可能性最大。

你说这个说法正确吗？请用程序验证。

2.7.21 用蒙特卡洛法计算椭圆面积。

2.7.22 用蒙特卡洛法计算球体的体积（球面方程：$X^2+Y^2+Z^2=1$）。

2.7.23 Buffon 丢针问题：平面上画着一些间隔为2a 的一组平行线。在平面上随机地投一枚长2l（l<a）的针，请模拟此丢针过程，并求出此针与一平行线相交的概率 P。

需要量 D_t	概　率
100台	0.25
200台	0.50
300台	0.25

2.7.24 某家商店经营收音机的销售业务，从过去的历史销售数据可以估计出市场每月对收音机需要量的概率分布，如右表所示。其中 D_t 表示第 t 个月的需求量。D_t 是商店不能控制的。商店能控制的只是进货数量 Q_t。

商店经理考虑到两种进货方案：

方案1：按上月的需要量来决定本月的进货数量，即令 $Q_t=D_{t-1}$；

方案2：按前两个月的需要量的平均数来决定本月的进货数量，即令 $Q_t=\frac{1}{2}(D_{t-1}+D_{t-2})$。

若卖出一台收音机获利20元，问哪一种进货方案获利大？

第三章　数　组

3.1　用数组组织数据

3.1.1　几个基本概念

1. 数组与下标变量

前面介绍了用 True BASIC 程序保存数据的两种形式：一种形式是用 DATA 语句把数据保存在内存特定的区域内。在需要对这些数据进行操作时，必须先用 READ 语句将数据读入变量。另一种形式是直接把数据保存在以变量名命名的存储单元中。用变量的形式保存的数据可以参加程序指定的运算。迄今为止在程序中使用的变量都是个别定义的：每个变量都有一个单独的名字，并且只能分别独立地存储一个数据元素。这种变量称为简单变量。简单变量之间没有内在的联系，例如变量 A 和变量 B 是互相独立的变量，它们并不表示从属或一定的顺序相邻的关系。

但是，现实世界中的事物之间往往是有联系的，例如，最常见的是数据之间具有顺序关系，如一个班 50 名学生，他们的年龄可以按某种顺序表示为 $A_1, A_2, \cdots, A_{50}$；一次物理实验中依次测得的 10 个电流值为 $I_1, I_2, \cdots I_{10}$；某商店中几种商品的价格可以依次表示为 $P_1, P_2, \cdots, P_n$ 等等。这些数据固然可以分别用简单变量来存储，如用 A，B，C，…分别存储第一、二、三、……位学生的年龄。这样存储既麻烦、不直观，又不能反映数据之间的本来顺序。为了能在程序中组织这类具有顺序关系的一组数据，高级语言都能提供一种“构造型数据结构”——数组。

数组(array)是一种由同类型数据组成的组合型数据结构。即它是用一个名字来命名一组同类型的数据。这个名字就称为数组名。也就是说，一个数组包含若干个数据，这些数据是按一定的顺序排列的。

数组中的每一个数据，称为数组元素。一个数组中的不同元素以下标(Subscript)来区别。下标代表数组元素在数组中的顺序。例如数组 SA 代表某班 30 个学生的年龄数据，并用 SA(1)存储第一名学生的年龄；SA(2)存储第二名学生的年龄；……；SA(30)存储第三十名学生的年龄。这里，SA 称为数组名，SA(1)，SA(2)，…，SA(30)，…称为数组元素或“带下标的变量”。简称下标变量。下标(如 1，2，…，30，…)是一个数值量，下标必须用圆括号括起来。即

SA(25)←—数组元素 (下标变量)
↑　　↑
数组名　下标

下标应当是一个整数。当程序员使用了非整数作下标时，系统将把它舍入成一个整数。

数组元素(下标变量)的使用与简单变量基本相同，即同一类型的简单变量和下标变量可以互相代替(只有循环变量例外，它只能用简单变量，不能用下标变量)。例如表达式

A * B+C,可以用I(1) * I(2)+I(3)代替,只要A和I(1),B和I(2),C和I(3)的值对应相等即可。

数组的引入,使我们可能方便地解决使用简单变量难于或无法处理的问题。

例3.1.1 处理学生成绩。

在数据处理中下面三个内容是常遇到的:(1) 输入数据;(2) 对数据排序或查找;(3) 按要求输出数据。

下面讨论用简单变量和数组分别处理上述三个内容。为简单起见,假设讨论的数据只有学生姓名与学生成绩两项。

(1) 输入数据

输入数据,就是把数据输入并保存到计算机内。在使用简单变量的情形下,数据一个一个地保存在各简单变量中。为了存储大量的数据,就必须命名一批变量。如对学生姓名与学生成绩这两种数据,就要分别命名 NAME1 $,SCORE1,NAME2 $,SCORE2,……,NAME30 $,SCORE30。当数据量很大(如100,1000,甚至更多)时,就要命名一大批变量,这是十分不方便的。如果用数组,可以不必定义大量的变量名,只须命名两个数组 SNAME $,SCORE 即可以了。用 SNAME(1)和 SCORE(1)代表第一个学生的名字和分数,SNAME $(N)和 SCORE(N)代表第N个学生的名字和分数。

此外,要解决如何把大量数据送入变量中。使用简单变量时,输入大量数据有两种方法:当数据已知时,可以先把数据用 DATA 语句存储在程序中,然后用 READ 语句将它们读入到相应的变量中去,如上述输入学生成绩的程序段可以写为

```
  ⋮
READ NAME1 $ ,SCORE1,NAME2 $ ,SCORE2,……
  ⋮
DATA 〈学生姓名 1,〉〈学生成绩 1〉,……
  ⋮
```

若数据事先不知道,在程序中则要使用 INPUT 语句,在程序运行时,输入数据。如上述学生成绩的输入程序段要写成

```
  ⋮
INPUT NAME1 $ ,SCORE1,NAME2 $ ,SCORE2,……
  ⋮
```

不论哪种方式,当数据量很大时,程序的书写工作量也是很大的。使用数组情形就好多了。如输入学生成绩的程序段可以写为

```
    ⋮
FOR I=1 to N
  read SNAME $ (I),SCORE(I)
NEXT I
    ⋮
DATA 〈学生姓名 1,〉〈学生成绩 1〉,……
    ⋮
```

或
```
    ⋮
```

```
FOR I=1 to N
    input SNAME$(I),SCORE(I)
NEXT I
  ⋮
```

显然,程序的书写工作量大大减小了,程序的可修改性提高了(例如学生数变动时,只须修改n),可读性也提高了。

(2) 排序与查找

用简单变量时,实现大量的数据的排序或从大量数据中查找某一数据,是很不方便的,有时不得不用手工方法进行。如 SCORE1=65,SCORE2=73,SCORE3=57,SCORE4=72,SCORE5=96,……,要将它们按分数由小到大的顺序排列,只能手工进行。采用数组后,由于可以用下标标识数组中的不同数据,因此可以用循环来处理数组,在每次循环中改变下标的值。例如,要从上述五名学生中,找出成绩为 72 分的学生的姓名,只需下面的一段程序:

```
LET N=5
FOR I=1 TO N
  IF SCORE(I)=72 THEN PRINT SNAME(I)
NEXT I
```

而要给出学习成绩大于 80 分的学生名单,只需使用下面一段程序:

```
LET N=5
FOR I=1 TO N
  IF SCORE(I)>80 THEN PRINT SNAME$(I)
NEXT I
```

学生人数改变时,只须改变 N 的值。

(3) 输出数据情形与输入相似,这里就不再讨论了。

总之,在处理大批数据时,用数组这种数据结构,比简单变量要方便得多。

2. 数组的类型与数组的命名

在例 3.1.1 中,为存储学生姓名,使用了数组 SNAME$。这是一个字符型数组,它的每个元素都是字符型数据。此外,还使用数值型数组 SCORE,它的每个元素都是数值型数据。

一般说来,数组是同类型数据的组合,它只能由同一类型的数据构造而成。按照 True BASIC 所提供的基本数据类型,数组被分为数值数组(numeric array)与字符数组(literal array 或 string array)两种类型。

数组的类型由所使用的数组名标识。数组的命名规则与简单变量相同。字符型数组名要以"$"结尾,如 SNAME$;否则就是数值型数组的数组名,如 SCORE。数组也应按"见名知义"的原则命名,以提高程序的可读性。

应当注意,在一个程序中不应当使用两个相同的符号名来命名不同的数组和简单变量。如程序段

```
LET A=5
FOR I=1 TO 5
  LET A(I)=A
```

NEXT I

程序中设数组名为 A，还有一个简单变量名也为 A，程序运行时，便会出现“类型错误”。

3．数组的维数

在例 3.1.1 中，使用的数组 NAME $ 和 SCORE 中的元素之间的顺序关系是一度（一维）的。即用以区别数组中的元素只需使用一个下标。这种数组称为一维数组。

除此之外，现实世界中还有许多事务之间具有二度乃至多度的顺序关系。如矩阵

$$B = \begin{bmatrix} B_{11} & B_{12} & \cdots & B_{1n} \\ B_{21} & B_{22} & \cdots & B_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ B_{m1} & B_{m2} & \cdots & B_{mn} \end{bmatrix}$$

中，每个元素的位置不仅有行的顺序（1，2，…，m）还有列的顺序（1，2，…，n）。当用一个数组存储该矩阵时，每个元素的位置都需要用行和列两个下标来描述，例如 C(2，1)表示 C 数据中第 2 行第 1 列的元素。C 数组是一个二维数组。图 3.1 是一个 3 * 2 * 3 的三维数组的示意图。它有三页、二行、三列，共可存储 3 * 2 * 3 个元素，每个元素的位置需要用三个下标来描述。为描述一个数组中的元素所需的下标个数，就是该数组的维数。

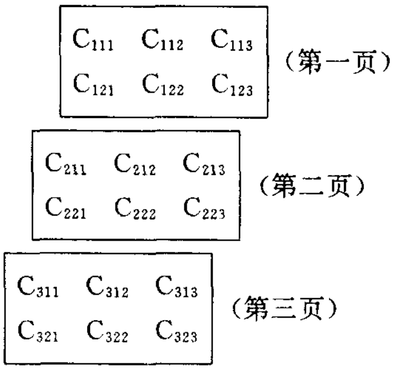


图 3.1

大维数的数组所占用的存储空间是惊人的。假设有一个 10 维数组，每维所含元素个数均为 10，则需要占用 10^{10} 个数据的存储空间。这对一般微机的内存容量来说，是难以实现的（如果一个数据用 4 个字节存储，则需要 4×10^{10} 个字节，即 40000 兆字节）。因此，节约内存的关键是节约使用数组，不要轻易定义大数组。

例 3.1.2 用二维数组处理学生成绩表。

在例 3.1.1 中，使用两个一维数组处理学生成绩表。在本例中用一个二维数组来处理学生成绩。

首先要定义一个二维数组。这个数组可以是一个 2 行 10 列的二维数组，也可以是一个 10 行 2 列的二维数组。在此二维数组中，要存储 10 名学生的姓名和他们的成绩。为了把成绩与姓名存在同一数组中，必须统一它们的数据类型。由于姓名只能是字符型数据，所以不得不把成绩也转换为字符型数据，把成绩作为字符型数据后，便不能直接对它进行任何算术运算（如求总成绩、平均成绩）了。

假定所使用的是一个 2 行 10 列的二维数组，那么，可用第一行的 10 个元素存学生姓名，用第二行的 10 个元素存这 10 名学生的成绩。向该数组中输入数据时，应使用二重循环，先输入 10 名学生姓名，再输入 10 名学生的成绩。程序段如下：

```
dim scores $(2,10)      ! 定义数组
data Zhang, Li,Wang,Zhao,Chen
data Guo,Liu,Ma,Fang,Gau
data 76,87,93,56,78,67,99,100,75,64
!
```

```
for i=1 to 2
  for j=1 to 10
    read scores $ (I,J)
  next j
next  i
```

如果知道了某学生的姓名，要找他的成绩，则应先找出该姓名在数组 scores $ 第一行中的列数，然后根据该列数在第二行中找出他的成绩。程序段如下：

```
print "input student name:";
input namex $
for i=1 to 10
  if scores $ (1,i)=namex $  then
    print namex $ ; "scores is:"; scores $ (2,i)
    exit for
  end if
next i
```

3.1.2 定义数组

数组是组织有序数据的构造型数据结构。它在内存中占用一片连续的存储单元。BASIC 数组一般采取"按行的顺序"存储方式，即先存储第一行元素，然后再存储第二行元素，……。例如数组

$$\begin{bmatrix} 3 & 8 & 4 & 5 \\ 2 & 6 & 1 & 3 \\ 4 & 5 & 3 & 2 \end{bmatrix}$$

在内存中存放的顺序如图 3.2 所示。

3
8
4
5
2
6
1
3
4
5
3
2

图 3.2

由于数组要在内存中占用连续的存储单元，为了让系统为它开辟并保留必要的存储空间，在使用一个数组之前，必须先对此数组进行定义。在 BASIC 中用 DIM (dimension 的缩写)语句定义。它的作用是向系统提供数组的下列信息：

① 数组名及数组类型(用数组名及类型符来描述)；

② 数组维数 (用下标的个数来描述)；

③ 各维的大小 (即每一维包含多少个元素。用每一维下标的上界和下界来描述)。

DIM 语句的格式为

> DIM〈数组 1〉(〈下界$_{11}$〉to〈上界$_{11}$〉[,〈下界$_{12}$〉to〈上界$_{12}$〉]……)[,〈数组 2〉(〈下界$_{21}$〉to〈上界$_{21}$〉[,〈下界$_{22}$〉to〈上界$_{22}$〉]……)]……

如语句

```
dim year (1900 to 1990), observer $ (1 to 10, -40 to 40)
```

dim 语句定义了两个数组:year 和 observer $ 。year 是一个一维的数组型数组，它共可以存储 1990－1900＋1＝91 个数值型数据。observer $ 是一个二维字符型数组，它有 10 行、81

列，共可存储 81×10=810 个字符型数据。

下面对 DIM 语句作几点说明：

(1) 尽量不要把维数定义得太大。True BASIC 允许一个数组最高为 10 维(也有些机器允许多于 10 维)，但是尽量不要定义高维数组，维数越多，所需的内存量增长越大，会造成存储空间的很大浪费。例如，本来想定义一个 array(10,20,30)，由于程序员未仔细核算，定义为 array(15,20,30)，即第一维所含元素由 10 改变为 15，则增加了 5×20×30=3000 个数据的存储空间，如果每个数据占 4 个字节，则浪费了 4×3000=1200≈12K 字节。维数越多可能造成浪费越大。

(2) 关于数组下标的上、下界

① 在 DIM 语句中定义数组某维的下标的上、下界，意味着定义了在本程序中该数组在该维上可以使用下标值的范围。例如，dim A (1 to 5)，定义 A 数组，可以使用的下标值为 1，2，3，4，5，即：数组元素有 A(1)，A(2)，A(3)，A(4)，A(5)。如果有：DIM B (−2 to 3)，定义了 B 数组，可使用的下标值为−2，−1，0，1，2，3，共 6 个值。数组元素有 B(−2)，B(−1)，B(0)，B(1)，B(2)。可知，在该维上所含元素个数为：(上界−下界+1)，或者说，在该维上可用的下标值有(上界−下界+1)个。

True BASIC 允许程序编制者根据需要定义下标的上、下界(而一般的 BASIC 版本和其它有些语言只允许定义下标的上界，不能定义下界，下界自动定为 1 或 0)。True BASIC 的规定对程序设计人员来说是比较方便的，更适应实际情况。只要知道在数组的某一维上所使用的下标值的实际范围，就可以直接定义出下标的上、下界。例如，想存储 1900～1999 年间每年的平均温度的数据并查询 1988 年的平均温度。可以定义一个温度数组。

```
dim temperature (1900 to 1999)
for i=1900 to 1999
  input temperature (i)
next i
print "temperature of 1988 is"; temperature (1988)
```

显然这是比较直观而方便的。如果不允许定义下标下界，则可以这样编写程序：

```
dim temperature (100)
for i=1 to 100
  input temperature (i)
next i
print temperature (1988−1900+1)
```

需要经过一定的转换，即 1900 年的温度放在 temperature (1)中，依此类推，1988 年的数据应放在数组中的第(1988−1900+1)个元素中。显然，这样表示是不方便，不直观的。

不言而喻，程序中使用的数组元素，其下标值应在定义的下标的上下界之间的范围内。

② 定义的下标的上、下界的值应是−1e9 到+1e9 范围内的整数。

③ 不允许使用变量或表达式定义下标的上、下界，例如，dim a(x to y，u to v/2)，将在编译时出错。

④ 允许用冒号代替“to”，例如上面述及的数组 temperatwre 也可用下面的形式定义：

dim temperature (1900 : 1990)

⑤ 当下界为 1 时，可省略“〈下界〉 to”部分，下面两个 dim 语句等价：

dim a(1 : 100, 1 : 10)

dim a(100,10)

⑥ 数组的上、下界允许在程序运行过程中用 MAT 语句进行调整。详见 3.2。

3.1.3 数组测试函数

True BASIC 提供了一些函数(如表 3.1 所示)，用以对数组的大小进行测试。这些测试函数都返回一个数值，这些数值可用于任何数值表达式中。

表 3.1

函　　数	含　　义	函　　数	含　　义
Size　　(A)	数组 A 中当前的元素数	Lbound　(A,n)	数组 A 的第 n 维的下界
Size　　(A,n)	数组 A 的第 n 维中的当前元素数	Ubound　(A)	一维数组 A 的上界
Lbound　(A)	一维数组 A 的下界	Ubound　(A,n)	数组 A 的第 n 维的上界

3.1.4 简单应用举例

程序不仅要说明(描述)算法，还要说明(描述)数据结构。在许多情况下，算法的描述要以数据结构为基础。数据结构的选择影响着算法的复杂性和整个的解题效果。通过例 3.1.1 和例 3.1.2 初步了解到，处理带有顺序关系的数据集时，采用数组与采用简单变量所采用的算法的复杂性大不相同。

数组是 BASIC 中一个基本的、十分重要的数据结构，应用得十分广泛。在本节中先举几个与数组有关的算法及应用实例。至于它的进一步用法，如内存排序、检索以及构成动态数据结构等，将在后面的章节中讨论。

1. 递推

在第二章中，已经初步介绍了递推的概念，下面再进一步加以讨论。

常常遇到这样的问题：在一个数的序列值中，下一项的值对前一项的值有某种依赖关系，求某项的值要从第一项起逐次推算而得。例如序列：1，2，4，8，16，32，…，256 中，后项的值是前项的值乘以 2。要求某一项，必须先用第一项乘以 2 求出第二项，然后再逐次求出第三项、第四项、第五项、……、直到最后一项。当然必须事先给定第一项的值。可以看出，第 n 项的值等于第(n－1)项的值乘以 2。即

$$\begin{cases} a_n = a_{n-1} \times 2 & (n > 1) \\ a_1 = 1 & (n = 1) \end{cases}$$

这种在规定的初始条件下，找出后项对前项的依赖关系的操作，称为“递推”(recurence)。表示这种关系的式子，称为递推关系式。上述的 $a_n = a_{n-1} \times 2$ 就是“递推关系式”，而 $a_1 = 1$ 称为“初始条件”或“边界条件”。

在实际问题中类似的情况很多。例如，银行本息的计算：假设 P_0 为本金，C 为利息，则年底本利和为 $P_1 = P_0 * (1 + C\%)$；第二年年底的本息为 $P_2 = P_1 * (1 + C\%)$；…。又如计算人口增长情况，都是以前一年的人口数为基数，计算下一年的人口数。

处理这类问题的理想方法是用归纳法写出一个直接由初始条件求第 n 项的公式来。但是在有些情况下直接写出求第 n 项的式子很难，写出一个递推关系式却比较容易，这时便可以采用递推技术：把求解一个问题的复杂过程，代之以由边界条件出发通过若干步简单的中间过程递推，最后达到最终解。这也是一种逐步细化思想。在 1.1.5 节中，曾用求解多项式和，小猴吃桃等问题，简单介绍过递推技术的应用。在用计算机递推求解的过程中，会产生一些中间结果，要在计算机中分别存储这些中间结果，并且要反映这些中间结果之间的顺序关系，就要使用一维数组。

例 3.1.3 兔子繁殖问题：意大利著名数学家 Fibonacci 曾提出一个有趣的问题：假定所有的一对（一雌一雄）新生的兔子都是在两个月后每月生下一对兔子，在没有死亡的情况下，由一对新生的兔开始，每月的兔子对数将组成一个数列：

1，1，2，3，5，8，13，21，34，…即第 1、2 个月有一对兔子，第 3 个月有 2 对兔子，第 4 个月有 3 对兔子，……。上面这个数列称为 Fibonacci 数列。

我们看到，要给出 Fibonacci 数的通项表示式是很困难的。但是，却很容易建立如下递推关系：

$$\begin{cases} F_n = F_{n-1} + F_{n-2} & (n \geqslant 3) \\ F_1 = F_2 = 1 & (\text{初始条件}) \end{cases}$$

按照这一递推关系，从 n＝3 开始，每递推一次，便可以得出一个 Fibonacci 数。当我们要得到 Fibonacci 数列中的每个数时，最理想的数据结构是一维数组。

下面是求前 12 个 Fibonacci 数的 True BASIC 程序：

```
! Program to evaluate the first 12 terms
! of Fibonacci sequence.
!
dim fib(12)
let fib(1),fib(2)=1
for term=3 to 12
  let fib(term)=fib(term-1)+fib(term-2)
next term
! print fibonacci series.
print "Fibonacci series are:"
for term=1 to 12
  print "Term"; term; "is: "; fib (term)
next term
end
```

```
Ok, run
  Fibonacci series are:
  Term 1    is: 1
  Term 2    is: 1
  Term 3    is: 2
  Term 4    is: 3
```

```
  Term 5    is: 5
  Term 6    is: 8
  Term 7    is: 13
  Term 8    is: 21
  Term 9    is: 34
  Term 10   is: 55
  Term 11   is: 89
  Term 12   is: 144
Ok.
```

看到上面的程序，读者一定会想起在 2.7.2 节中介绍过的迭代算法。在设计计算机算法时，递推与迭代都可以用重复结构来实现，所以有些计算机的书籍中并不区分这两个概念，甚至不去区分递推、迭代和重复这三个概念。但严格地说，它们是有区别的，首先看看下面用迭代法求 Fibonacci 数列的程序：

```
! printe to evalnate the first 12 terms
! of Fibonacci sequence
!
let fib1, fib2=1
print "Fibonacci series are:"
print "The first two terms are 1,1"
for term=3 to 12
  let fib=fib1+fib2
  print "Term"; term; "is: "; fib
  let fib1=fib2
  fet fib2=fib
next term
end
```

```
Ok. run
  Fibonacci series are:
  The first two terms are 1,1
  Term 3    is: 2
  Term 4    is: 3
  Term 5    is: 5
  Term 6    is: 8
  Term 7    is: 13
  Term 8    is: 21
  Term 9    is: 34
  Term 10   is: 55
  Term 11   is: 89
  Term 12   is: 144
Ok.
```

我们看到，在迭代程序中至少存在着一个迭代变量，程序每重复一次，便能直接或间接地由它的旧值推得一个新值，上例中 fib，fib1，fib2 是迭代变量。而递推过程中不存在这样的迭代变量，由 fib(1)，fib(2)推出 fib(3)，由 fib(2)，fib(3)推出 fib(4)，……它们都不是迭代变量，因为它们的值没有被新值取代。

2. 检索

检索(search)就是从一组数据中找出所需的具有某种特征的数据项。它是数据处理中应用很广的一种操作。最容易理解的一种检索方法是顺序检索。这是一种穷举法。其基本思想是对所存储的数据从第一项开始，依次与所要检索的数据进行比较，直到找到该数据或将全部元素都找完还没有找到该数据为止。

设有 10 个数已存在数组 A 中，A 已定义为有 10 个元素，要找的数据为 x，这一检索过程用 True BASIC 描述如下：

```
  dim A(10)
  input x
  for i=1 to 10
    if A(i)=x then exit for
  next i
* if i>10 then
    print "this data has not found."
  else
    print "the data has been found.";
    print "it is A("; i; ")."
  end if
```

程序中打“*”的条件语句是为了测试循环是在找到 x 后结束的，还是没有找到 x 而结束的。

一般情况下所要找的数据是随机的，如果要找的数据正好就是数组中第一个数据时，只须找一次便可以找到；如果它是数组中最后一个数据时，就要查找 n 次(设数组有 n 个元素)，所以平均查找次数为

$$m=\frac{1}{n}(1+2+\cdots+n)=\frac{1}{2}(n+1)$$

显然，数据量越大，需要查找的平均次数也越多。

若被检索的是一组有序数据，就有可能用一种效率较高的方法检索。例如，有一批数据已按大小排列好。

$$a_1<a_2<\cdots\cdots<a_n$$

分别存储在数组 A(1)，A(2)，…，A(N)中，现在要对该数组进行检索，看给定的数据 x 是否在此数列中。可以用下面的方法：

在 1 到 n 中间选一个正整数 k，用 k 把原来的有序数列分为三个有序数列：

I1：A(1)，A(2)，…，A(k－1)

I2：A(k)

I3：A(k＋1)，A(k＋2)，…，A(N)

然后，用 A(k)与 x 比较，若 x=A(k)，查找结束；若 x<A(k)，则用同样的方法把序列 A(1)，A(2)，…，A(k)分成三个序列；若 x>A(k)，也用同样的方法把序列 A(k+1)，A(k+2)，…A(N)分成三个序列，直到找到 x 或得到“x 找不到”的结论为止。

这是一种应用分治策略的解题思想。当 $k=\frac{1}{2}$时，称为二分检索法(bin-srch)。图 3.3 为二分检索法的算法描述。其中变量 F 是“是否找到”的标志。设 F 的初值为 0，当找到 x 后置 F=1。根据 F 的值便可以确定循环是由于找到 x(F=1)结束的，还是由于对数据序列查找完了还找不到结束的(F=0)。

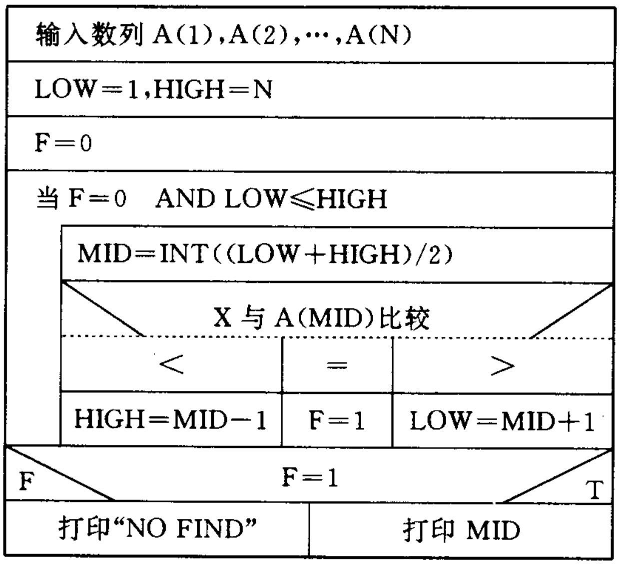


图 3.3 BINSRCH 算法

例 3.1.4 一个二分查找程序

```
! Program of bin-search
!
data 0,1,2,3,4,5,6,7,8,9
dim A(10)
let n=10
for i=1 to n
  read A(i)
next i
input prompt "X=" : x
let low=1
let high=n
let f=0
let k=0.5
do while f=0 and low<=high
   let mid=int((low+high) * k)
   if x=A(mid)then
     let f=1
   else if x<A(mid) then
     let high=mid-1
   else if x>A(mid) then
     let low=mid+1
   end if
loop
if f=0 then
```

```
  print "No find"
else
  print "x is A("; mid;")"
end if
end
```

```
Ok. run
X=13↙
No find
Ok. run
X=2↙
X is A(3)
Ok. run
X=10↙
No find
Ok. run
X=9↙
X is A(10)
Ok. run
X=0↙
X is A(1)
Ok.
```

用二分法查找,最好的情况是查找一次就找到,即 x=A(N/2)。设 $N=2^m$,则最坏的情况要查找 m+1 次。显然数据最多时,用二分法查找比用逐次法查找要经济得多。

在二分法查找的基础上发展起来的查找算法还有 0.618 查找与 Fibonacci 数列查找。

只使用简单变量存储数据无法使用二分检索法。读者由此也可以看出数据结构对解题效率和成败有很大影响。

使用二分检索法的前提是数据序列必须先排好序。关于排序方法,将在 3.3 中介绍。

3. 数据表格处理

由例 3.1.1 与例 3.1.2 可以看出,用处理数据表格,非常直观又很方便。下面再举一个例子。

例 3.1.5 库存数据处理:某仓库库存商品统计情况如表 3.2 所示,各种商品的价格如表 3.3 所示。下面介绍如何利用数组对这些数据表进行处理。

(1) 存储数据报表:用二维数组 stock 表 3.2 的数据,用一维数组 price 存储表 3.3 的数据,程序段如下:

```
!
data 8506.30,3000,4500,6000
data 8512.15,7000,2000,3000
data 8607.03,2000,1500,5000
data 8612.20,3500,2500,8000
data 8607.12,9000.2000,7500
```

表 3.2

年 月 日 \ 库存量 \ 商品	C1	C2	C3
85.6. 30	3000	45000	6000
85.12.15	7000	2000	3000
86.7. 3	2000	1500	5000
86.12.20	3500	2500	8000
87.6. 12	9000	2000	7500

表 3.3

商品	价格
C_1	150
C_3	120
C_3	80

```
data 150,120,80
!
! Read in the data
dim stock(5,0 to 3),price (3)
for i=1 to 5
  for j=0 to 3
    read stock (i,j)
  next j
next i
for i=1 to 3
  read price(i)
next i
```

在上述程序中，日期本来不是数值型数据，但是为了把它们与库存量一起存放在数值型数组 stock 中，故把它们写成数值形式。

(2) 查找某一数据：如果要查找 1986 年 7 月 3 日商品 C_2 的库存量，可用下面的程序段：

```
for i=1 to 5
  if stock(i,0)=8607.03 then
    print stock(i,2)
    exit for
  end if
next i
```

(3) 统计库存金额：如要统计 1986 年 12 月 30 日的总库存值，可用下面的程序段：

```
for i=1 to 5
  if stock (i,0)=8612.02 then
    print stock(i,1) * price(1)+stock(i,2) * price(2)+stock(i,3) * price(3)
    exit for
  end if
next i
```

用户可以根据需要把这几个程序段组合连接起来，写出一个完整的程序。

4. 矩阵运算

二维数组与数学中的矩阵密切相关，矩阵运算也是二维数组最常见的应用之一。下面举例说明。

例 3.1.6 矩阵相加：两个 m×n 的矩阵 A 和 B 相加，就是将它的对应位置元素相加，得到一个新的 m×n 的矩阵 C。算法如下。

S1：输入矩阵 A 和 B；

S2：分别对每个同位置元素求和，得到和矩阵的同位置元素；

S3：打印和矩阵。

这是一个"抽象算法"，要对这个抽象算法细化，首先要决定应选用什么样的数据结构。显然，合理的数据结构是用数组 A,B,C 存储矩阵 A,B,C。决定了数据结构后，便可以针对特定的数据结构对上述算法进行细化了。这个算法很简单，下面直接给出程序如下：

```
! Program of matrix of additioen
!
data 2,3,4,1
data 3,6,2,3
data 1,5,8,3
let m=3
let n=2
dim a(3,2), b(3,2), c(3,2)
! input matrix data
for line=1 to m
  for col=1 to n
    read a(line,col),b(line,col)
  next col
next line
!
! matrix addition
for line=1 to m
for col=1 to n
    let c(line,col)=a(line,col)+b(line,col)
  next col
next line
!
! print added matrix
for line=1 to m
  for col=1 to n
    print c(line,col);
  next col
  print
next line
end
```

```
Ok, run
  5  5
  9  5
  6  11
Ok
```

例 3.1.7 矩阵相乘：设 A 矩阵为 m×l(m 行，l 例)、B 矩阵为 l×n(l 行，n 列)，两个矩阵相乘，要求 A 矩阵的列数(l)与 B 矩阵的行数(l)相同。得到的乘积矩阵为 m×n，即 m 行 n 列。

$$\underset{m\times l}{A}\cdot\underset{l\times n}{B}=\underset{m\times n}{C}$$

C 矩阵中的元素 $C_{ij}=\sum_{k=1}^{l}a_{ik}b_{kj}$，其中 $i=1,2,\cdots,m$；$j=1,2,\cdots,n$。例如，

$$C=A\cdot B=\begin{bmatrix}1&2\\5&3\\3&1\\2&2\end{bmatrix}\cdot\begin{bmatrix}-1&-2&3\\1&2&1\end{bmatrix}$$

$$=\begin{bmatrix}1\times(-1)+2\times1&1\times(-2)+2\times2&1\times3+2\times1\\5\times(-1)+3\times1&5\times(-2)+3\times2&5\times3+3\times1\\3\times(-1)+1\times1&3\times(-2)+1\times2&3\times3+1\times1\\2\times(-1)+2\times1&2\times(-2)+2\times2&2\times3+2\times1\end{bmatrix}=\begin{bmatrix}1&2&5\\-2&-4&18\\-2&-4&10\\0&0&8\end{bmatrix}$$

其中：$c_{23}=\sum_{k=1}^{2}a_{2k}\cdot b_{k3}=a_{21}\cdot b_{13}+a_{22}\cdot b_{23}$

矩阵乘积的乘法可以表示如下：

S1：输入矩阵 A 和 B；

S2：计算矩阵 C=A×B；

S3：打印矩阵 C。

S1 与 S3 前面已作过介绍，这里只对 S2 的细化过程加以说明。根据矩阵乘的定义，首先把 S2 细化为图 3.4 所示算法，再对 S2.1 细化得到图 3.5 所示算法。

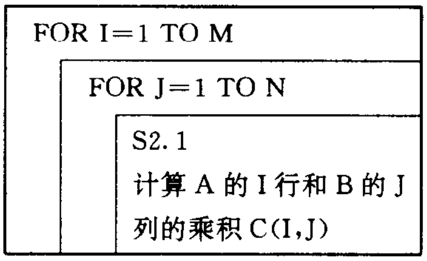


图 3.4

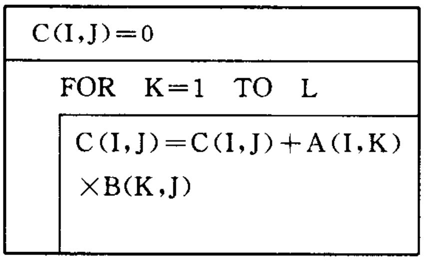


图 3.5

由图 3.4 及图 3.5 所示算法，可以得到如下程序段：

```
for i=1 to m
  for j=1 to n
    let C(i,j)=0
    for k=1 to l
        let C(i,j)=C(i,j)+A(i,k) * B(k,j)
    next k
  next j
next i
```

注意存储矩阵 A 的数组为 A(m,l)，存储矩阵 B 的数组为 B(l,n)，存储矩阵 C 的数组

为 C(m,n)。

例 3.1.8 矩阵转置。将 m×n 的矩阵 A 的行和列互换，得到 n×m 的矩阵 AT，AT 称为 A 的转置矩阵。例如

$$A=\begin{bmatrix}1 & 2\\ 3 & 4\\ 5 & 6\end{bmatrix},\ AT=\begin{bmatrix}1 & 3 & 5\\ 2 & 4 & 6\end{bmatrix}$$

这个算法很简单，直接写出转置处理的程序段如下，其余部分请读者自己完成。

```
for i=1 to n
  for j=1 to m
    let AT(i,j)=A(j,i)
  next j
next i
```

例 3.1.9 用高斯消元法解线性方程组。线性方程组的一般形式为

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 & (3.1.1)\\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 & (3.1.2)\\ \cdots\cdots & \cdots\\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n & (3.1.n)\end{cases}$$

将系数组成 A 矩阵，常数 $b_1,b_2,\cdots,b_n$ 组成 B 矩阵，变元 $x_1,\cdots,x_n$ 组成 X 矩阵，得

$$A=\begin{bmatrix}a_{11} & a_{12} & \cdots & a_{1n}\\ a_{21} & a_{22} & \cdots & a_{2n}\\ & \cdots\cdots & & \\ a_{n1} & a_{n2} & \cdots & a_{nn}\end{bmatrix},\ X=\begin{bmatrix}x_1\\ x_2\\ \vdots\\ x_n\end{bmatrix},\ B=\begin{bmatrix}b_1'\\ b_2\\ \vdots\\ b_n\end{bmatrix}$$

高斯消元法的基本思想是首先把原方程组变换为三角方程组，即令

$$A'=\begin{bmatrix}a_{11} & a_{12} & a_{13} & \cdots & a_{1n}\\ 0 & a_{22}' & a_{23}' & \cdots & a_{2n}'\\ 0 & 0 & a_{33}' & \cdots & a_{3n}'\\ \cdots & \cdots & & & \\ 0 & 0 & 0 & \cdots & a_{nn}'\end{bmatrix},\quad B'=\begin{bmatrix}b_1'\\ b_2'\\ b_3'\\ \vdots\\ b_n'\end{bmatrix}$$

然后使 A′矩阵的对角线为“1”即得

$$A''=\begin{bmatrix}1 & a_{12}'' & a_{13} & \cdots & a_{1n}''\\ 0 & 1 & a_{23}'' & \cdots & a_{2n}''\\ 0 & 0 & 1 & \cdots & a_{3n}''\\ \cdots & \cdots & & & \\ 0 & 0 & 0 & \cdots & 1\end{bmatrix},\ B''=\begin{bmatrix}b_1''\\ b_2''\\ b_3''\\ \vdots\\ b_n''\end{bmatrix}$$

这样由最后一个方程可以得到 $x_n=b_n''$，然后逐一代回前面的方程，便可以求得原方程组的全部解。即用高斯消元法解线性方程组的算法为

S1：输入方程组系数矩阵；

S2：将原方程组三角化；

S3：将三角方程组的对角线系数化为“1”；

S4：由第 n 个方程求出 x_n，再逐次回代，求得原方程组的解；

S5：打印结果。

上述算法的实现过程实际上是对矩阵 A 与向量 B 进行运算变换的过程。显然，用二维数组 A(1∶n,1∶n)表示系数矩阵 A，用向量 B(1∶n)表示向量 B 是很合适的。

确定了数据结构后，便可以在所选用的数据结构的基础上对抽象算法细化了(实际上，确定数据结构本身是对抽象算法的细化过程中重要的一步)。

(1) 对 S1 细化。S1 即输入数组 A 与 B 中各元素，这部分很简单，请读者自己完成。

(2) 对 S2 细化。

S2 的功能是把二维数组 A 变换为三角形。行列式有一个性质：将行列式某一行(列)的所有元素同乘以数 R 后加到另一行(列)对应位置的元素上，行列式不变。根据这一性质，若用－A(K,I)/A(I,I)乘数组 A 与 B 中的第 I 行各元素，然后加到第 K 行中各元素上，便可以使 A(K,I)＝0，当从 K＝I+1 到 K＝N 重复上述算法时，便可以将数组 A 中从第 I+1 行到 N 行中的第 I 列变为零。即

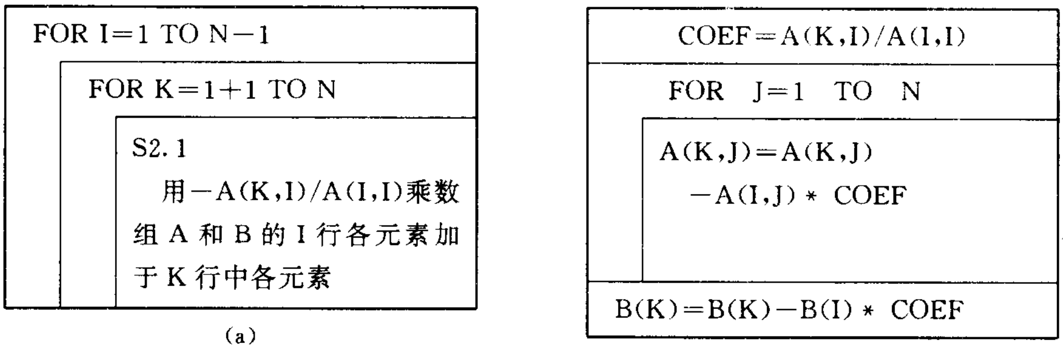
当 I＝1 时，将 A 数组中的第 2 到 N 行的第 1 列都变为零；

当 I＝2 时，将 A 数组中的第 3 到 N 行的第 2 列都变为零；

⋮

当 I＝N－1 时，将 A 数组中的第 N 行中的第 N－1 列变为零。

这样便得到了一个使数组 A 变为三角形的算法，如图 3.6(a)所示。其中 S2.1 是用－A(K,I)/A(I,I)乘数组 A 和 B 的 I 行各元素，加到 K 行各元素。即用－A(K,I)/A(I,I)乘 A(I,1)，A(I,2)，…，A(I,N)，B(I)加到 A(K,1)，A(K,2)，…，A(K,N)，B(K)上。但是由于 A 数组中第 I 行的 1～(I－1)列均为零，所以只要用－A(K,I)/A(I,I)乘 A(I,I)，A(I,I+1)，…，A(I,N)，B(N)，分别加到 A(K,I)，A(K,I+1)，…，A(K,N)，B(K)上即可，算法如图 3.6(b)所示。



(a)　　　　(b)

图 3.6

到此为止，似乎对 S2 的细化完成了。但是，我们忽视了这样一个问题：如果在上述算法中出现 A(I,I)＝0，则将无法计算 COEF。解决的办法是：从第 I+1 行到第 N 行，逐一找第 I 列不为零的 A(L,I)，作为主元，并将数组 A 与 B 的 L 行与 I 行全部元素对调。若找完第 N 行后还找不到，则说明该方程组无解。按此思想对图 3.6 所示算法进行修正，得到图 3.7 所示算法。

到此为止,对 S2 细化完毕。

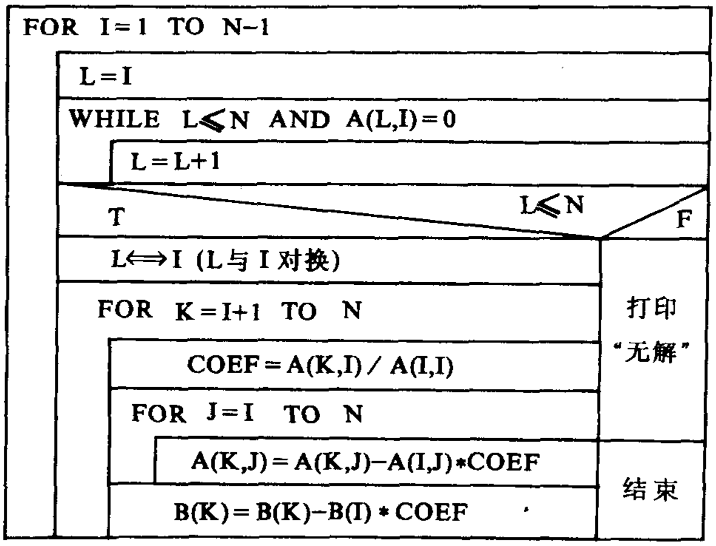


图 3.7 注:L 与 I 对换,即第 L 行与第 I 行对换

(3) 对 S3 细化。

S3 是将化成三角形的数组 A 的对角线元素变为 1。即用 A(I,I)去除第 I 行中各元素。算法如图 3.8 所示。

(4) 对 S4 细化。

执行完 S2,S3 后,得到方程组

$$\begin{cases} x_1 + a''_{12}\ x_2 + a''_{13}\ x_3 + \cdots + a''_{1n}\ x_n = b''_1 \\ \qquad x_2 + a''_{33}\ x_3 + \cdots + a''_{2n}\ x_n = b''_2 \\ \qquad\qquad \cdots\cdots\cdots\cdots \\ \qquad\qquad x_{n-1} + a''_{(n-1)n}\ x_n = b''_{n-1} \\ \qquad\qquad\qquad x_n = b''_n \end{cases}$$

由上面最后一个方程得 $x_n=b''_n$。依次代回,可逐次求得 $x_{n-1}, x_{n-2}, \cdots, x_2, x_1$ 的值。

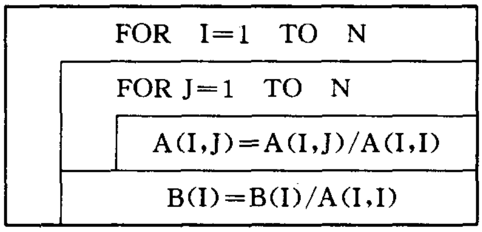


图 3.8

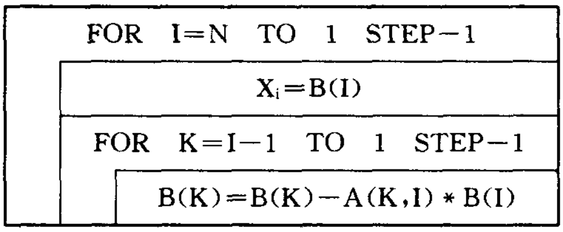


图 3.9

算法如图 3.9 所示。其思路是从第 N 个方程开始,求出 x_n,并逐次找出 $x_{n-1}, x_{n-2}, \cdots\cdots$,每找到一个 x_I 后,将前面第(I-1)到第 1 个方程中的 x_i 都消去,即使方程组消去一个变元。

(5) 对 S5 细化。

将上述各子算法连接起来并加以优化,便可以得到用高斯消元法求解线性方程组的算法,再用 True BASIC 代码代替,便可得到 True BASIC 程序。这些工作请读者自己完成。

在此题中，也可以用一个 n×(n+1)的数组存放方程组的增广矩阵

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ a_{21} & a_{22} & \cdots & a_{2n} & b_2 \\ & & \cdots\cdots & & \\ a_{n1} & a_{n2} & \cdots & a_{nn} & b_n \end{bmatrix}$$

对增广矩阵进行变换，使其化为对角线元素为“1”的上三角形，再逐次回代。利用这种形式存储，在计算机解题中比较方便。详细算法及程序请读者自己完成。

3.2 数组的输入与输出

数组数据的传送可以通过两种方式进行：

1. 一种是通过使用下标确定数组元素，逐个地对数组元素进行存取。上面介绍的都属于这种方式。

前面已经指出，数组元素与简单变量的地位与作用完全相同，凡是可以使用简单变量的地方(除了作循环变量外)都可以使用数组元素。反之亦然。具体地说，可以象使用简单变量那样，用 INPUT 语句从键盘上向一组数据元素传送数据；用 READ 语句把程序数据区中数据传送给一组数组元素；用 LET 语句把一个表达式的值传送(赋给)一个或几个数组元素；用 PRINT 语句把若干个数组元素的值传送到输出通道——显示器或打印机。

2. 另一种方式是把数组作为一个整体进行数据传送。

用上面一种办法逐个传送数组元素数据不方便，需要程序设计者十分细心地逐一处理数组中各元素，稍有不慎就会有遗漏或出错。事实上，在许多情形下。需要对数组进行整体操作。例如，从键盘上将若干个数据输入到计算机中赋给一个数组的每个元素；或将一个数组中每一个元素打印出来，等等。在这种情况下，如果用上面的方式处理，使用 INPUT，READ/DATA，及 PRINT 语句输入输出时，就要使用重复结构。

True BASIC 能够用一个 MAT 语句进行一个数组的整体操作，从而简化了程序，提高了程序设计的效率和程序的简明性、易读性。

MAT 语句提供的操作分两大类：

1. 数组的输入/输出传送；

2. 数组的运算与赋值。

这一节只介绍数组的输入/输出传送。数组的运算与赋值放在下一节介绍。

3.2.1 MAT PRINT 语句

MAT PRINT 语句可以向输出通道——显示器或打印机传送整个的数组的数据。其一般格式为

```
MAT PRINT 〈数组 1〉;〈数组 2〉;…
```

或

```
MAT PRINT 〈数组 1〉,〈数组 2〉…
```

用一个 MAT 语句可以处理若干个数组。

1. 在 MAT PRITN 语句中，“,”或“;”的作用是：使打印出来的元素之间按“,”或“;”的

规定输出。

例 3.2.1

```
! Print two arrays.
!
dim A(10),B(10)
for i=1 to 10
  let A(i),B(i)=i
next i
★ mat print A,B;
★ mat print A;B
end
```

```
Ok. run
  1          2          3       4      5
  6          7          8       9      10
  1  2  3  4  5  6  7  8  9  10
  1  2  3  4  5  6  7  8  9  10
  1          2          3       4      5
  6          7          8       9      10
Ok.
```

注意：MAT PRINT 语句中的“,”或“;”只影响它前面的数组的输出格式，而不影响其后的数组的输出。若一个 MAT 语句最后一个数组是按“,”格式输出时，则其后“,”可以省略。

2. True BASIC 在对数组进行输出操作时，按数组元素在内存中排列的顺序输出。即“按行输出”，对二维数组而言，是输出完一行元素后再输出下一行的元素。每当一行的元素输出结束时，就换一行接着输出。

例 3.2.2

```
! Print two arrays.
!
dim A(3,5),B(3,5,3)
for i=1 to 3
for j=1 to 5
  let A(i,j)=i * 10+j
  for k=1 to 3
    let B(i,j,k)=i * 100+j * 10+k
  next k,
next j
next i
mat print A; B;
end
```

```
Ok. run
  11   12   13   14   15
  21   22   23   24   25
  31   32   33   34   35

  111   112   113
  121   122   123
  131   132   133
  141   142   143
  151   152   153

  211   212   213
  221   222   223
  231   232   233
  241   242   243
  251   252   253

  311   312   313
  321   322   323
  331   332   333
  341   342   343
  351   352   353
```

True BASIC 在输出完每个数组后，空一行。在输出三维或多维数组时，输出一“页”(二维)后也空一行。

3. MAT PRITN 语句不同于 PRINT 语句，它无计算功能，不能进行两个数组的运算。

```
dim A(2,3), B(2,3)
for i=1 to 2
  for j=1 to 3
    read A(i,j), B(i,j)
  next j
next i
mat print A+B
data 1,2,1,2,1,2
data 3,4,3,4,3,4
end
```

打入 RUN 命令后，系统检查程序，将光标停在★行中的“+”处，在背景窗口显示出错信息：“Doesn't belong here.”。

3.2.2 MAT INPUT 语句

1. MAT INPUT 语句的基本用法

使用一条 MAT INPUT 语句一次便可以从键盘上读入一个完整的数组。如

```
dim A(2,3)
mat input A
```

相当于下面的程序段：

```
dim A(2,3)
for i=1 to 2
  for j=1 to 3
    input A(i,j)
  next j
next i
```

MAT INPUT 语句的一般形式是

MAT INPUT 〈数组 1〉〈数组 2〉，……

以及

MAT INPUT PROMPT "〈字符串表达式〉"：〈数组 1〉,〈数组 2〉,……

说明如下：

(1) 输入数据时，应按 MAT INPUT 语句中所列出的数组名，以从左到右的顺序，一个数组一个数组的输入数据。如 MAT INPUT A、B、C，则要先给 A 的各元素输入数据，再给 B 的各元素输入数据，最后给 C 的各元素输入数据。

(2) 对每一个数组来说，应按数组元素在计算机内存中排列顺序输入数据。如对程序

```
dim A(2,3)
mat input A
end
```

从键盘上应按如下次序输入数据给 A 的各元素：A(1,1)，A(1,2)，A(1,3)，A(2,1)，A(2,2)，A(2,3)。

(3) 同一个数组中的数据应在一个数据行上输入，所输入的各元素的值应以逗号分隔。如果一个数组的元素太多，不能在一行中输入完，则应在行尾加逗号，之后回车，在下一行输入剩下元素的值。不同的数组的数据不应在一个数据行上输入，而应由不同的数据行输入。

2. 用 MAT INPUT 语句调整数组大小

每个数组在它被使用前必须用一个 DIM 语句定义。这个 DIM 语句确定了数组的维数和每一维中的上、下界。在程序运行过程中，还可以根据需要对某个数组的大小进行调整。只要不改变数组的维数，可以重新定义每一维所包含元素的个数，可以使数组变大或变小。也可以用 MAT INPUT 语句重新定义(即调整)数组大小，在 MAT INPUT 语句中允许使用表达式或变量定义数组的上、下界。如

```
mat input A(10)
mat input B(2 to 5, 3 to 7)
mat input C(i to j, k to k+5)
```

其中的 i、j、k 应事先赋值。

例 3.2.3 比较下面的两个程序。

程序 1：

```
! MAT PRINT without redimensioning.
dim A(2,2)            ! square array
mat input A           ! get all 4 elements
print "Thanks."
end
```

```
Ok. run
? 1,2,3↙
Too few input items. Please add more.
? 4↙
Thanks.
Ok.
```

程序 2：

```
! MAT PRINT with redimensioning to fixed bounds.
!
dim A(2,2)                ! 2 * 2 arrays
mat input A(1,2)          ! make it 1 * 2 and input it
print "Thanks."
end
Ok. run
? 1,3,5,7↙
Too many input items. Excess ignored.
Thanks.
Ok.
```

MAT INPUT 语句还可以使用一个"(?)"，对一维数组的上界进行调整.这个数组所包含的元素事先并不确定，根据实际输入的数据个数而定。例如执行语句

mat input a(?)

时，如用户输入 5 个数据，则 a 数组就被定义为包含 5 个元素，即上界为 5。

例 3.2.4

```
! MAT INPUT with vairiable lenth redimensioning.
!
dim A(100)
mat input A(?)
print Ubound(a); "items."
end
```

```
Ok. run
? 1,2,3,4,5,↙
? 6,7,8↙
```

```
 8 items.
Ok.
```

当从键盘上打入一行数据后用一个逗号结尾时，表示还要输入数据。直到最后一行用非逗号结尾时，输入数据才结束。

3.2.3 MAT READ 语句

当数组元素的值事先已确定时，便可以用 READ/DATA 语句将各元素的值赋给数组。MAT READ 语句能够一次读入一个或多个数组的所需的值。如

```
dim A(2,3), B$(5)
mat read A,B
```

相当于下面的程序段：

```
dim A(2,3), B$(5)
for i=1 to 2
   for j=1 to 3
      read A(i,j)
   next j
next i
for i=1 to 5
   read B$(i)
next i
```

MAT READ 语句的一般形式是

MAT READ 〈数组 1〉, 〈数组 2〉, ……

MAT READ 语句读入数据的顺序与 MAT INPUT 语句相同。它也是按从左到右的顺序给各数组一个一个地赋值；对每个数组来说也是按计算机内存中数据元素排列的顺序给数组元素赋值。

用 MAT READ 不必等待从键盘的人工输入，因而执行效率比 MAT INPUT 高。也可以用 MAT READ 语句对数组进行调整，方法与 MAT INPUT 语句相似，但是不能使用“A(?)”的形式。

例 3.2.5

```
! Set up by reading months and days of week.
!
dim month $ (100), weekday $ (7)
mat read month $(12), weekday $
data January, February, March, April,May,June
data July,August,September,October,November,December
data Monday,Tuesday,Wednesday,Thursday,Friday
data Satuday,Sunday
mat print month $ ,weekday $
```

```
end
```

```
Ok. run
January      February     March        April        May
June         July         August       September    October
November     December

Monday       Tuesday      Wednesday    Thursday     Friday
Satuday      Sunday

Ok.
```

3.3 数组赋值与运算

3.3.1 MAT 赋值语句

MAT 赋值语句允许用常量、变量、表达式或其它数组给一个数组赋值。

1. 用表达式给数组赋值。

格式：

MAT〈数组名〉=〈变量或常量〉

或

MAT〈数组名〉=(〈表达式〉)

这种赋值方式将使数组中的每个元素都获得与表达式(或常量或变量)相同的值。

例 3.3.1

```
! MAT assignment of each element.
!
dim A(2,3,4),B$(2,6),C(2,2,2)
★mat A=(2*Pi) (用表达式赋值)
★mat C=PI (用常量赋值)
★mat B$=("Shan"&"xi") (用常量赋值)
mat print A;B$;C;
end
```

```
Ok. run
 6.28319  6.28319  6.28319  6.28319
 6.28319  6.28319  6.28319  6.28319
 6.28319  6.28319  6.28319  6.28319

 6.28319  6.28319  6.28319  6.28319
 6.28319  6.28319  6.28319  6.28319
```

```
6.28319   6.28319   6.28319   6.28319

ShanxiShanxiShanxiShanxiShanxiShanxi
ShanxiShanxiShanxiShanxiShanxiShanxi

3.14159   3.14159
3.14159   3.14159

3.14159   3.14159
3.14159   3.14159
Ok.
```

说明如下：

(1) 在 MAT 赋值语句中，赋值号“=”左边的数组类型应与赋值号右边的常量、变量或表达式的类型一致。

(2) 如果表达式包含运算符号，则应把它用圆括号括起来。如果表达式只由一个变量或常量组成，则不必用括号。

(3) 赋值号左边只能是一个数组名，而不能是多个数组名。即使给两个以上数组赋以同一常数时，也要分别使用 MAT 赋值语句。如要给 A，B 两个数组中的全部元素都赋值以 5 时，应写成

```
MAT A=5
MAT B=5
```

而不能写成

```
MAT A,B=5
```

这一点与 LET 语句是不同的。

2. 用数组给数组赋值

MAT 语句能把一个完整的数组赋给另一个数组变量。这时 MAT 语句的一般形式是

MAT 〈数组 1〉=〈数组 2〉

要求数组 1 与数组 2 具有相同的维数。执行 MAT 语句后，便能把数组 2 各元素的值传送给数组 1，并改变数组 1 的上界(下界不变)使数组 1 与数组 2 的大小一致。

例 3.3.2

```
  ! Matrix assignments.
  !
  dim A(5:6,5:6),B(30,20)                ! A has 4 elements, B 600.
★mat B=3                                 ! Every element in B is 3.
★mat A=B                                 ! Now A is same size as B.
 print, "Lower bound to", "Upper bound", "Size"     ! Prove it.
 print "A", Lbound(A,1), Ubound(A,1)
 print,Lbound(A,2),Ubound(A,2),Size(A)
 print
 print "B",Lbound(B,1),Ubound(B,1)
```

```
print,Lbound(B,2),Ubound(b,2),Size(b)
end
```

```
Ok. run

            Lower bound to      Upper bound     Size
A           5                   34
            5                   24              600

B           1                   30
            1                   20              600
Ok.
```

可以看出，执行 mat A=B 语句后，数组 A 的第一维的上界变为 34，使其与数组 B 的第一维有相同的元素个数；数组 A 的第二维的上界变为 24，使其与数组 B 的第二维有相同的元素个数。即 A 变成 A(5：34,5：24)，A 数组第一维所含元素变为 30，第二维所含元素变为 20。但数组 A 的两个下界都没改变。在执行 MAT A=B 后，A 数组变成与 B 数组一样大小(每维中所含元素相同)。

3.3.2 数组加

如果两个数组，它们的维数和大小相同，可以进行相加(减)。这两个数组相加(减)的过程，是将对应位置上的元素相加(减)，把和(差)再放到目标数组的相应位置上。

图 3.10(a)、(b)分别是两个三维数组相加、减的运算情形示意图。

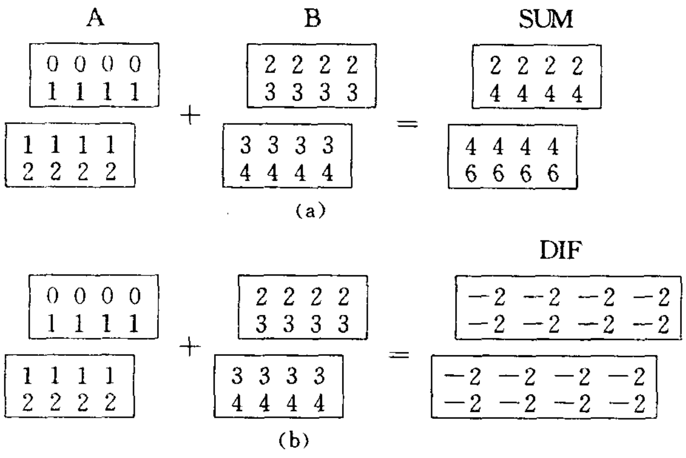


图 3.10

MAT 赋值语句能够进行上述的数组相加(减)，格式如下：

MAT SUM=A+B

MAT DIF=A−B

注意：A，B 的维数与大小必须相同。如果 A 是一个 2×3×4 的数组，B 也应是一个 2×3×4 的数组。即第一要维数相同，第二要各维所包含的元素数相同(但并不要求各维的下标

的上、下界也相同)

例 3.3.3 实现图 3.10 所示数组加、减的程序:

```
! Array sums and differeces.
!
dim A(2,2,4), B(2:3,3:4,4), SUM(2,2,2),DIF(2,2,2)
mat read A,B
mat SUM=A+B
mat DIF=A-B
mat print A; B; SUM; DIF;
data 0,0,0,0,1,1,1,1
data 1,1,1,1,2,2,2,2
data 2,2,2,2,3,3,3,3
data 3,3,3,3,4,4,4,4
end
```

```
Ok. run
   0  0  0  0 ┐
   1  1  1  1 │
   1  1  1  1 ├数组 A
   2  2  2  2 ┘
   2  2  2  2 ┐
   3  3  3  3 │
   3  3  3  3 ├数组 B
   4  4  4  4 ┘
   2  2  2  2 ┐
   4  4  4  4 │
   4  4  4  4 ├数组 SUM
   6  6  6  6 ┘
  -2  -2  -2  -2 ┐
  -2  -2  -2  -2 │
  -2  -2  -2  -2 ├数组 DIF
  -2  -2  -2  -2 ┘
Ok.
```

MAT 赋值语句执行数组加后,将结果赋值给一个同样维数的数组变量,同时可以对该数组的大小进行调整(每一维的下界不变,只调整上界)。

3.3.3 矩阵乘

只有一维或二维数组才有可能进行数组乘法。如果 A 是一个 L×M 的数组,B 则应是一个 M×N 的数组,A×B 所得的数组是一个 L×N 的数组。图 3.11 是两个数组相乘的例子。

$$\begin{array}{|c|c|}\hline 1 & 4 \\ \hline 2 & 5 \\ \hline 3 & 6 \\ \hline \end{array} \times \begin{array}{|c|c|c|c|}\hline 1 & 3 & 5 & 7 \\ \hline 2 & 4 & 6 & 8 \\ \hline \end{array} = \begin{array}{|c|c|c|c|}\hline 9 & 19 & 29 & 39 \\ \hline 12 & 26 & 40 & 54 \\ \hline 15 & 33 & 51 & 69 \\ \hline \end{array}$$

图 3.11

MAT 赋值语句能够对两个符合矩阵乘要求的矩阵进行乘运算，并把结果赋值给一个目标数组。赋值时对目标数组进行相应调整。

例 3.3.4 实现图 3.11 所示矩阵乘的程序。

```
! Matrix multiplication.
!
dim A(3,2),B(2,4)
mat read A,B              ! A is 3 * 2, B is 2 * 4
mat A=A * B               ! Multiply them
mat print A;              ! Result is 3 * 4
data 1,4,2,5,3,6          ! Data for A
data 1,3,5,7,2,4,6,8      ! Data for B
end
```

```
Ok. run
  9    19    29    39
  12   26    40    54
  15   33    51    69

Ok.
```

3.3.4 数值量乘数组

用 MAT 语句可以实现用一个数(常数、简单数值变量或数值表达式)去乘一个数组，其结果是数组的每个元素都用同一个数去乘。如

```
mat A=n * B
```

B 是数组名，n 是数值变量。注意，不能写成 MAT A=B * n。

如果要乘的数不是简单数值变量或常数，则必须用圆括号括起来。如

```
mat A=(n+m) * B
mat B=(-1) * C
```

3.3.5 内部数组常数

True BASIC 规定了一些专用的数组值，这些值可以用来向数组赋值。这些内部数组见表 3.2。

表 3.2

内部数组值	含　义
Con	一个所有元素皆为1的数组。当它赋给某一数组时，它具有同目标数组相同的维数和大小。
Con(x,y,…)	一个所有元素皆为1的数组，它是一个x×y×…数组。
Idn	一个单位矩阵(一个正方形矩阵，其对角线上所有元素皆为1，其余元素皆为0)，在为一个目标数组赋值时　它自动具有同目标数组相同的大小。
Idn(x)	一个x×x的单位矩阵。
Nul $	一个所有元素皆为空字符串的字符串数组，它具有同目标数组相同的维数和大小。
Nul $ (x,y,…)	一个所有元素皆为空的字符串数组，它是一个x×y×…数组。
Zer	一个所有元素皆为0的数组，它具有同目标数组相同的维数和大小。
Zer(x,y,…)	一个所有元素皆为0的数组，它是一个x×y×…数组。

内部数组值共有以上四组，每一组都有两种不同的格式：一种是不带下标界限的，另一种是带下标界限的。如果在向一个目标数组赋值时，内部数组值不带任何下标界限，则目标数组保持它原来的大小。如

```
dim a(10,10)
    ⋮
mat a=con
```

则a数组保持为10×10的大小，每个数组元素的值为1。

如果数组值是指定下标界限的，则目标数组将按此来调整大小。如上述的a数组改用赋值语句。

```
mat a=con(5,5)
```

则a数组改为5×5大小的数组。数组各元素值为1。

例3.3.5

```
!
dim A(2,5), B(3,3), C $ (2,5),D(3,5)
mat A=Con
mat B=Idn
mat C $ =Nul $
mat D=Zer
print "A"
mat print A;
print "B"
mat print B;
print "C $ "
mat print C $
print "D"
```

```
mat print D;
end
```

```
Ok. run
A
  1    1    1    1    1
  1    1    1    1    1

B
  1    0    0
  0    1    0
  0    0    1

C$



D
  0    0    0    0    0
  0    0    0    0    0
  0    0    0    0    0
Ok.
```

例 3.3.6

```
!
dim A(2,5),B(3,3),C$(2,5),D(3,5)
mat A=Con(3,6)
mat B=Idn(5)
mat C$=Nul$(3,5)
mat D=Zer(3,8)
print "A"
mat print A;
print "B"
mat print B;
print "C$"
mat print C$;
print "D"
mat print D;
end
```

```
Ok. run
A
  1    1    1    1    1    1
  1    1    1    1    1    1
```

```
 1    1    1    1    1    1

B
 1    0    0    0    0
 0    1    0    0    0
 0    0    1    0    0
 0    0    0    1    0
 0    0    0    0    1

C$

D
 0    0    0    0    0    0    0    0
 0    0    0    0    0    0    0    0
 0    0    0    0    0    0    0    0
Ok.
```

请读者分析比较例 3.3.5 与例 3.3.6 所产生的结果。

3.3.6 矩阵函数

True BASIC 提供以下四个矩阵函数：

1. DET(x)和 DET

Det(x)用以返回方阵 x 的行列式|x|的值。如果 x 是奇异的，则 Det(x)=0。

如果 DET 函数不带自变量，则返回上一次用 INV 函数求逆矩阵的行列式的值。如果未曾对任何矩阵求过逆，则 DET 返回零。

2. DOT(x,y)

Dot(x,y)返回两个向量 x 和 y 的点积。要求这两个向量必须有相同的元素数。其结果为一个数值，是 x 和 y 的内积。

3. INV(x)

Inv(x)返回方阵 x 的逆矩阵，如果 x 是一个奇异矩阵，或近似于一个奇异矩阵，则 True BASIC 给出一个出错信息。在对矩阵求逆之后最好使用 Det 函数检验矩阵是否近似于奇异。如果 Det≈0，则求逆的结果可能是无意义的。

4. TRN(x)

函数 Trn(x)返回 x 的转置矩阵。x 必须是二维数组。一个 M×N 的矩阵 x 的转置是一个 N×M 的矩阵 R，其中 R(i,j)=X(j,i)。

3.4 排　　序

按照预先规定的准则(如升序或降序等)，把数据有次序地排列起来的操作称为"排序"或"分类"(Sorting)。排序可以提高查找的效率(如对已排序数据进行二分查找。已排序的数据还常常隐含某些特殊的信息，如按升学考试成绩排序的学生名单中，隐含了按成绩录取的

顺序以及可录取的分数线等信息。

排序是数据处理中的一项重要功能。随着近十几年来数据处理规模的迅速增长，在各个国家中计算机用于处理排序所花的总时间越来越多。据统计，一些商用计算机系统中，15%到70%的CPU时间用在排序上。因此，研究排序算法是计算机科学中的一项重要课题。

目前已经设计出许多排序算法。在众多的排序算法中，要简单地判断哪种算法最优、可以普遍采用，是不现实的，一种排序算法的效率，常常与数据的类型、规模乃至计算机系统的特性有关。一般说来，对于给定的应用环境，可以从以下三个方面来评价排序算法：

1. 算法的时间开销。“数据比较”与“数据移动”是组成排序算法的主要操作。一个排序算法中所包含的比较次数和数据移动次数决定了该排序算法的时间效率。

2. 算法的空间开销。在排序过程中，除要存放待排序的数据外，还要使用一些辅助存储空间。排序算法空间效率可用所使用的辅助存储空间的数量来评价。

3. 程序设计的难易程度。

下面介绍几种典型的排序算法。这些排序算法除其实用价值外，是给初学算法设计者以启迪和训练的好范例。

在介绍这些排序算法之前，还需要指出两点：

(1) 排序之前的数据是一个序列，排序之后的数据又形成一个新的序列。在BASIC程序中，处理排序的合理数据结构是数组。

(2) 为叙述方便，后面介绍的排序算法都是针对递增排序设计的。如果数据要递减排序，其算法思想是完全相同的。

3.4.1 选择排序

选择排序(selection sort)的基本思想是，先在未排序序列中选一个最小元素，作为已排序子序列，然后再重复地从未排序子序列中选取一个最小元素，把它加入已排序的数列，作为已排序子数列的最后一个元素，直到把未排序子数列中的元素处理完为止。

简单选择排序(simple selection sort)是一种最容易理解的选择排序方法。为了说明简单选择排序方法的思想，先看如何用简单选择法对六个人按个子高矮进行排队：6个人任意排好队，先让左起第一个人站出队外，然后让站在队外的人与站在队内的人(左起第二到第六人)依次比个子，若站在队内的人个子矮，则让他与站在队外的人对换。一趟比较完后，站在队外的人必然比站在队内的所有人的个子都要矮。这时让他站回左起第一个位置去，作为已排好高矮的子队列中的第一名。然后再让左起第二名站出来，让他们按上述规则与第三到第六名比较高矮，比完后便选出从第二名到第六名五人中的最矮的人，让他站到左起第二位，作为已排好高矮的子队列中的第二名(此时就是最后一名)。如此反复，选过五趟矮个子后，便可以把这6个人全部按个子高矮排好队。

若6个人用其身高代表，上述排序过程如图3.12所示。方框中的数相当于上述“站在队外的人”。箭头表示数据传送操作。带下划线者为已排好的队列。在第一趟比较时，先将第一个数172放在框内，作为与第2～第6个数比较的基础，然后再与第二个数168比较，由于168<172，因此，二者对换，172变成第二个数，168成为框内的数，再用框内的数168与后面第3～第6个数比较，比到第五个数时，由于163<168，因此，再发生互换，框内的数变成163。这就是6个数中的最小数，将它作为已排序的子数列的第一个数。现在还剩下5个

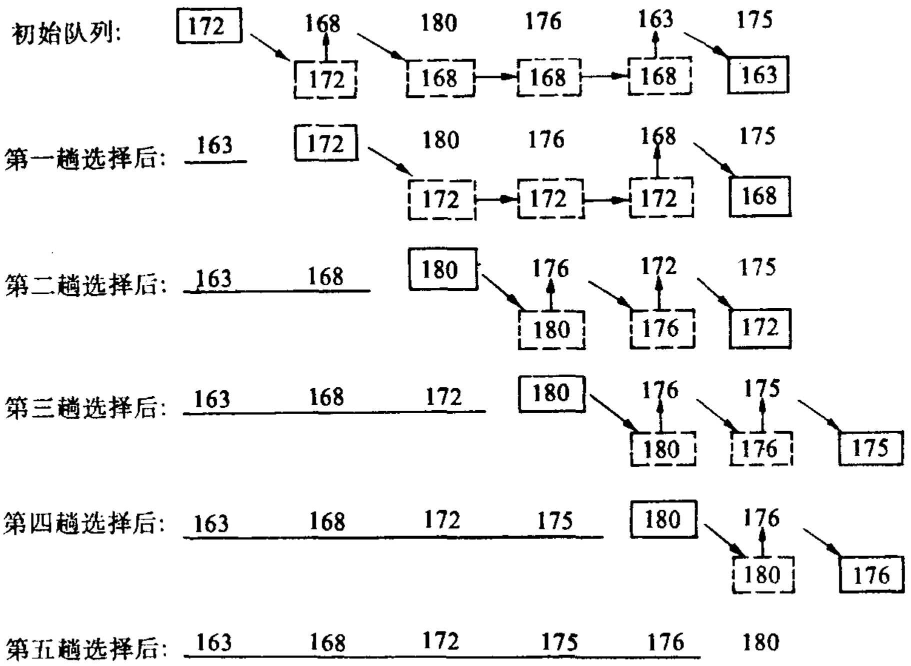


图 3.12

数未排序,在第二趟比较中,将第二个数 172 放在框内,再与第 3～第 6 个数比较,方法同前。比完五趟后,6 个数已排好序。

一般情况下,当有 N 个人时要进行 N－1 趟。每一趟都要让出队的人(位置设为 I)与其下一人(位置为 I+1)直到最后一人(位置为 N)逐一进行比较,并按一定规则换位。用数组 A(1：N)存储这 N 个人的身高,便可以得到图 3.13 所示的排序算法。其中变量 X 是为交换 A(I)与 A(J)内容而设置的中间变量。外循环次数为比较的趟数(N－1 趟),内循环的次数是在比较某一趟时,所需的比较次数。

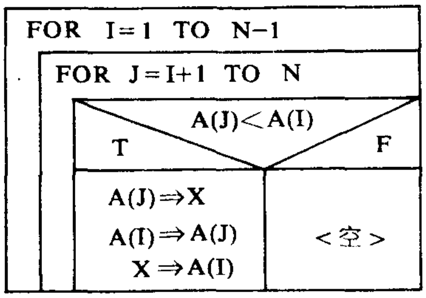


图 3.13

图 3.13 所示的算法,数据交换的次数比较多。由于简单选择排序只要求每一趟选出一个较小元素排在已排好序的子序列的最后,所以有可能做到每一趟最多只"交换"一次。解决的方法是,设置一个指针(即一个变量),首先让它指向未排好序的子数列的第一元素(即该元素的顺序号——下标的值放入此变量中),然后让该元素与其后各元素逐一比较,在比较过程中使指针总是指向较小的元素(即使最小元素的顺序号——下标值放在此变量中)。经过一趟比较后,该变量中的值便是未排好序子数列中的最小元素的下标值。或者说,指针指向最小元素。若指针指向的不是当前未排好序子数列中的第一个元素,则要把指针指向的元素与当前未排好序子数列的第一个元素交换,让它加入已排好序子数列作为最后一个元素。按这一思想,对图 3.13 修改得图 3.14 所示算法。这就是常用的简单选择排序算法。按照这一算法不难写出 True BASIC 程序如下:

```
! Simple selection sort.
!
```

```
dim a(10)
print "Input your data items:"
mat input a(? )
let n=Ubound(a)
for i=1 to n-1
  let pointer=i
  for j=i+1 to n
    if a(j)<a(pointer) then let pointer=j
  next j
  if i< >pointer then
    let x=a(i)
    let a(i)=a(pointer)
    let a(pointer)=x
  end if
next i
mat print a;
end
```

```
Ok. run
Input your data items:
? 172,168,180,176,163,175↙
 163 168 172 175 176 180
Ok.
```

用简单选择法排序时，比较次数与数据原来的顺序无关。即第一趟总要比较 n－1 次，第二趟总要比较 n－2 次，……，第 n－1 趟要比较 1 次。总的比较次数为

$$\sum_{i=1}^{n-1}(n-i)=\sum_{i=1}^{n}(i-1)=\frac{n}{2}(n-1)$$

对已排好序的数据序列来说，比较的次数同样需要 $\frac{n}{2}(n-1)$ 次，但不必移动(交换)任何数据，而原来是逆序时，需要移动数据 3(n－1)次。(因为要交换 n－1 次，每交换一次数据要用三个赋值语句，即移动 3 次，故总共 3(n－1)次)。

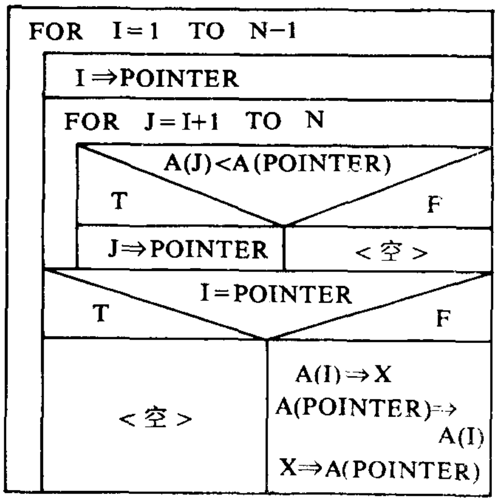


图 3.14

3.4.2 插入排序

插入排序(insertion sort)是将待排序的数据按一定的规则逐一插入到已排序数列中的合适位置处，直到将全部数据都插入为止。插入的规则不同，便形成了不同的插入排序方法。

1. 直接插入排序：直接插入排序(straight insertion sort)先以未排序数列的第一个元

素作为已排序子数列，然后从原来的第二个元素起，将各元素逐一插入到已排序子数列中合适的位置，直到把全部元素都插入为止。图 3.15 是用直接插入法对 6 个人按身高高矮进行排序的示意图。

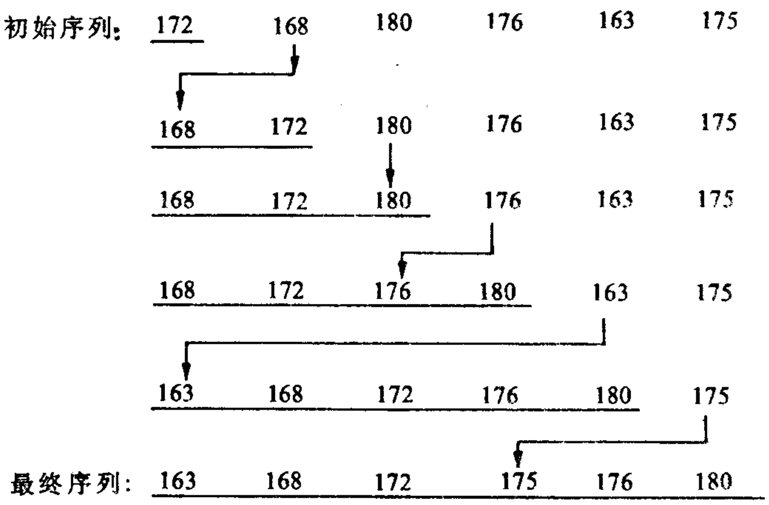


图 3.15

下面讨论如何设计用直接插入法对保存在数组 A(N)中的 N 个数据进行排序的算法。

按照直接插入法，首先要把 A(1)作为已排序子数列。然后逐一将 A(2)，A(3)，…，A(N)插入到已排序子数列中。每插入一个元素都要执行如下两个操作：

(1) 在已排序子数列中找一个合适位置 J，使 A(J)<A(I)，而 A(J+1)>A(I)。A(I)为 A(2)，A(3)，…，A(N)中将要插入到已排序子数列中的一个元素。例如，图 3.15 中的第三行，已排好序的子数列为 168，172，180。今要将 A(4)即 176 插入到上述子数列中，可找到 J=2，从图可知，A(2)<A(4)，A(3)>A(4)，(A(2)=172，A(3)=180)。

(2) 将 A(I)插入到 A(J)之后。

根据上述分析，可以得到图 3.16 所示算法。下面对它细化。

S1：将 A(1)作为已排序子数列
S2：FOR I=2 TO N
在子数列的位置[1，I−1]间找一个合适位置 J，将 A(I)插入到 A(J)之后

图 3.16

找插入位置比较容易，只需用穷举法对已排序子序列(即 A(1)，A(2)，…，A(I−1))按顺序逐一测试便可。测试可以从 A(1)开始往后进行，也可以从 A(I−1)开始往前进行。

找到 J 后，要将 A(I)插入到 A(J)之后，就要先将 A(J+1)，A(J+2)，…，A(I−1)各移后一个位置。由于要把 A(I)加入到已排序子数列中，所以可先把 A(I)的值装入到一个辅助单元 X 中，再把 A(I−1)的值⇒A(I)，A(I−2)的值⇒A(I−1)，…，A(J+1)的值⇒A(J+2)，于是 A(J+1)单元便可以用来存放 X 中值，即原来的 A(I)的值。这一过程如图 3.17

所示。

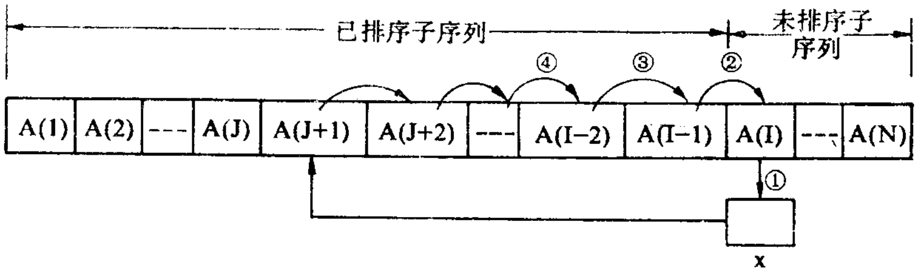


图 3.17

由上述分析可知，把“找插入位置”与“后移”同时进行，比较合理。即先将 A(I)⇒X，然后从 J=I−1 开始往前测试 A(J)，若 A(J)>X，则将 A(J)⇒A(J+1)中，再测试前面的一个元素，直到找到一个位置 J 为止。再将 X⇒A(J+1)，即插入到位置 J+1 中。

在下述两种情况下算是找到了 J：

① X 开始大于等于 A(J)；

② J=0，也就是 I=1(因为 J=I−1)，此时要将 X 插入到数列第一个位置。

根据上述分析，可以得到图 3.18 所示用直接插入法对数列 A(1)，A(2)，…，A(N)排序的算法。图 3.18 中已将图 3.16 中 S1 略去，因为 S2 中已隐含了 S1。

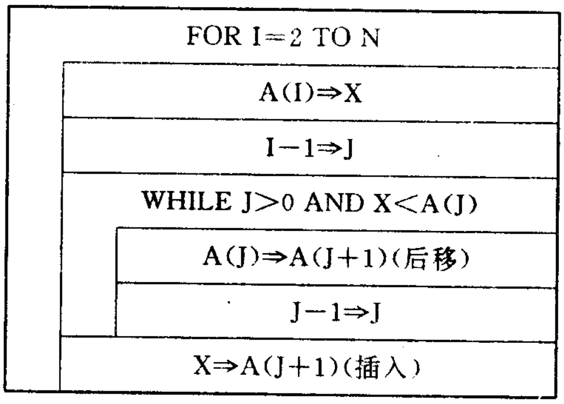


图 3.18

按图 3.18 所示算法可以写出如下 True BASIC 程序：

```
! Straight insertion sort.
!
dim a(10)
print "Input your data items:"
mat input a(? )
let n=Ubound(a)
for i=2 to n
  let x=a(i)
  let j=i−1
  do while i>0 and x<a(j)
    let a(j+1)=a(j)
    let j=j−1
  loop
  let a(j+1)=x
next i
mat print a;
end
```

```
Ok. run
Input your data items:
? 172,168,180,176,163,175↙
  163 168 172 175 176 180
Ok.
```

直接插入排序的算法算洁，容易实现。经过分析知道：完成一次排序所需的最小比较次数为 $C_{min}=n-1\approx n$，最大比较次数为 $C_{max}=\sum_{i=n}^{n} i=\frac{1}{2}(n-1)(n+2)\approx\frac{n^2}{2}$；数据的最小移动次数为 $M_{min}=2(n+1)\approx 2n$，最大移动次数为 $M_{max}=\sum_{i=2}^{n}(i+1)\approx\frac{n^2}{2}$。可见当 n 较小时，直接插入排序是一种较好的排序方法，而当 n 很大时，它的效率较低。

2. 二分插入排序：在直接插入排序中是采用逐次检索法顺序地查找插入位置。但是，已排序子数列已经有序；为了提高查找效率可以用二分检索法查找插入位置。这样可以把总的比较次数减少到 $n\cdot\log_2 n$ 次。数据移动次数与直接插入法相同。这种插入排序方法称为二分插入排序(binarry insertion sort)。

关于二分插入排序法的算法这里不作详细介绍。有兴趣者，可自己思考并完成。

3. Shell 排序(希尔排序)：Shell 排序(Shell's method)是由 D. L. Shell 于 1959 年提出的一种高效率的插入排序方法。

分析直接插入排序过程可以看出，它在数列局部有序时，就有一些数据不进行插入，而当数据量较小时，n^2 与 n 相差不多，所以效率较高。而 Shell 排序就是基于这两点对直接插入排序进行改进。它的基本思想是把一个较大的未排序的数列分割为若干较小的子数列，分别进行直接插入排序，并逐步扩大整个数列的有序性，直到完全有序。

那么，如何对原数列进行分割并逐步扩大其有序性呢？Shell 的作法是：先取一个整数增量 $d_1<n$，把原数列中所有相隔 d_1 的数据作为一组(可知一共分成 d_1 个组)，在各组内进行排序；然后取 $d_2<d_1$，重新分组并排序；……直到 $d_i=1$ 将所有数据放在一起进行一次排序为止。请看图 3.19。开始时取 $d_1=4$，将第 1,5,9 个数据为一组，同样，将第 2,6 个数据为一

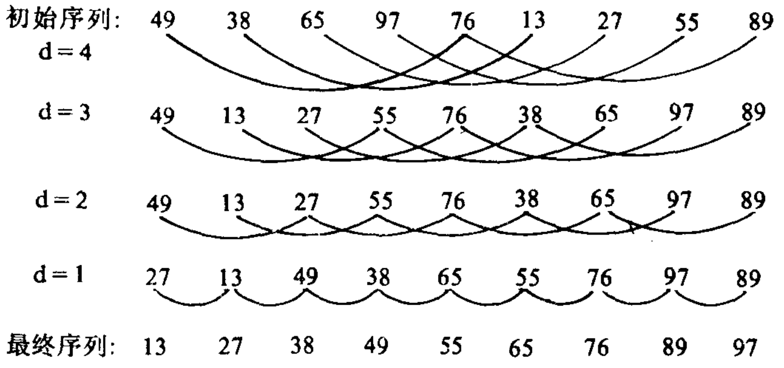


图 3.19

组，……，在每组内排序，结果是第 2 和第 6 个数(38 和 13)对换，以下类推。采用这种方法，开始时 d 的取值较大，每组中的数据较少，排序时比较次数少，且需移动的数据是跳跃地而

不是一步步地移动，排序效率较高；后来增量 d 值减小，各子数列内的记录数增多，但它们的有序程度也已有所提高，比较及移动次数仍较小。所以 Shell 排序能在整个过程中保持较高的排序速度。图 3.19 是对一个数据序列用 Shell 法排序的过程。

下面讨论如何设计对随机存储在 A(N)中的 N 个数据进行 Shell 排序的算法。

Shell 排序也称缩小增量排序(diminishing increment sort)。从缩小增量这一点上，把 Shell 排序抽象为如下过程：先取一个初始增量 D，不断用缩小的增量对原序列分割，并分别对 D 个子数列进行插入排序，这 D 个子数列中的每一个都是由相距 D 的元素组成的。如此不断减小 D 的值，一直到 D=1。这一过程可以用图 3.20(a)所示算法来描述。

S1：给定初始增量 D
WHILE D≥1
　S2：分别对 D 个子数列进行插入排序
　S3：取下一个 D

(a)

S1：D=INT(N/2)
WHILE D≥1
　S2：分别对各元素相距 D 的 D 个子数列进行插入排序
　S3：D=INT(D/2)

(b)

图 3.20

为了对 S1 与 S3 细化，应解决如何取增量数列(D1,D2,D3,…,1)的问题。增量数列的函数是决定 Shell 法效率的关键，Shell 最早提出的选法是，$D_1=int(N/2)$，$D_2=int(D_1/2)$，…；后来 knuth 提出选择 $D_{i+1}=int(D_{i-1})$；有人认为 D_i 取奇数好，也有人认为 D_i 之间互为素数为好，哪种方案最优，至今尚未得到证明。这里我们选 Shell 当初的方法。于是把 3.20(a)细化为图 3.20(b)。

下面再对 S2 细化。解决如何分别对各元素间相距 D 的 D 个子数列进行插入排序。考虑可以有两种方法：一种方法是先对子数列 A(1)，A(D+1)，A(2D+1)，A(3D+1)，…进行插入排序，然后对子数列 A(2)，A(D+2)，A(2D+2)，A(3D+2)，…进行插入排序，……最后对子数列 A(D)，A(2D)，A(3D)，…进行插入排序。用这种方法，S2 可以用图 3.21 所示算法来描述。借助图 3.18 所示直接插入算法，对它细化，并代入图 3.20(b)，可以得到图 3.22 所示 Shell 排序算法。根据这一算法可以很容易地写出用 Shell 法对 N=10 数据排序的程序及运行结果：

FOR G=1 TO D
　对 A(G)，A(D+G)，A(2D+G)，A(3D+G)，…子数列进行插入排序

图 3.21

```
! Shell's method to sorting:
!
dim A(10)
print "Input your data items."
mat input A(? )
let N=Ubound (A)
```

```
let D=Int(N/2)
do while D>=1
   for G=1 to D
      for I=D+G to N step D
         let X=A(I)
         let J=I-D
         do while J>0 and X<A(J)
            let A(J+D)=A(J)
            let J=J-D
         loop
         let A(J+D)=X
      next I
   next G
   let D=Int(D/2)
loop
mat print A
end
```

```
Ok. run
Input your data items:
? 49, 38, 65, 97, 76, 13, 27, 55, 89↙
  13  27  38  49  55  65  76  89  97

Ok.
```

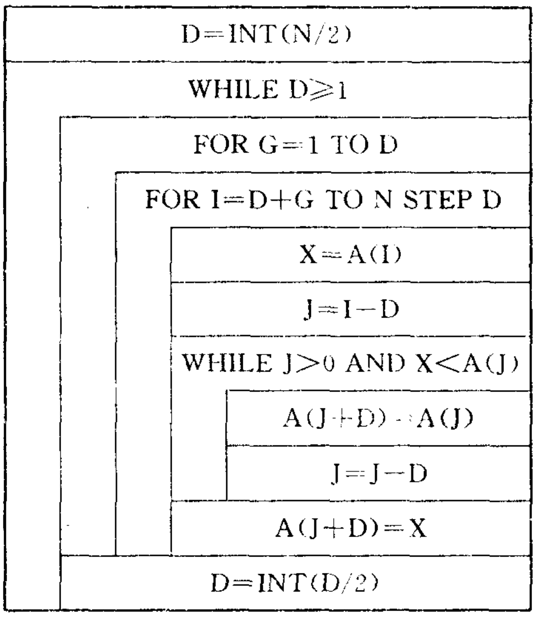


图 3.22

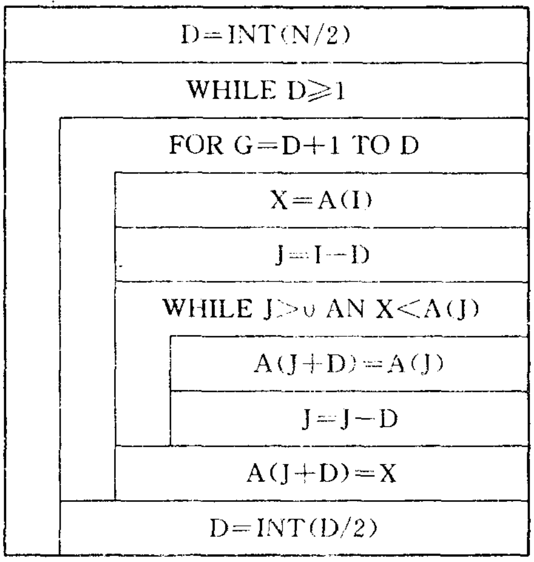


图 3.23

另一种对各元素相距为 D 的 D 个子数列进行插入排序的方法不是分 D 次进行，而是对

D 个子数列同时进行。具体做法是：由于各子数列的第二个元素分别是 D+1 到 2D，第三个元素分别是 2D+1 到 3D，第四个元素分别是 3D+1 到 4D，……，可以先把 D+1 到 2D 的 D 个元素分别插入到前面与其相距 D 的已排序子数列中，再把 2D+1 到 3D 的 D 个元素分别插入到前面各元素都相距 D 的 D 个子数列中，……，这实际上就是把从 D+1 到 N 的(N－D)个元素依次插入到前面的 D 个已排序子数列中去。根据这一思想，可以得到图 3.23 所示的另一种 Shell 排序算法。按照这一算法写出 True BASIC 程序的工作留给读者去完成。

Shell 排序一般说来比直接插入排序要快。有人证明，当 n 在一定范围时，其平均比较次数和平均移动次数均为 $n^{1.3}$左右；当 $n \to \infty$时，可减少到 $n \cdot (\log_2 n)^2$ 次。

3.4.3 交换排序

交换排序的基本思想，是按一定的规则(随具体方法而异)比较待排序数列中两个数据，若是逆序，则交换它们的位置，否则比较下一对，直到将全部序列排好序为止。常用的一种交换排序方法是起泡排序(bubble sort)。

设有 N 个数据依次放在数组 A(N)中，用起泡法对这 N 个数据进行递增排序的过程：先比较 A(1)与 A(2)，若逆序(即 A(1)>A(2))，就交换 A(1)与 A(2)；接着比较 A(2)(新的)与 A(3)，若逆序就交换；……；依次进行，直到将 A(N－1)与 A(N)比较交换完，才算完成了第一轮比较交换。然后依同样规则再从头进行第二轮比较交换，……；一直到进行完第 N－1 轮比较交换为止。图 3.24 为用起泡法对一组数据序列 7,5,3,9,1 排序过程示意图。

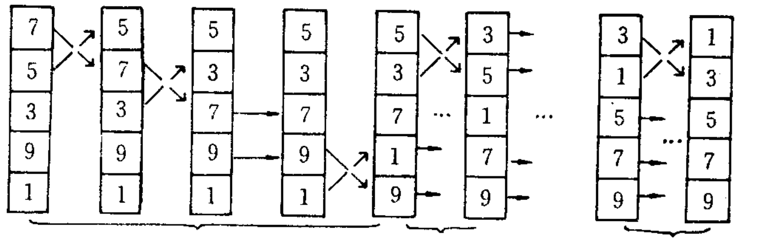


图 3.24

我们看到，在起泡排序过程中小元素好似水中的汽泡一样逐轮向上飘浮(如在图 3.24 中，每经一轮比较交换，数字“1”便升起一个位置)，而且每经一轮比较交换都有一个最大的元素沉淀到底部(如图 3.24 中第一轮把 9 沉到底部，第二轮把 7 沉到底部，……)。通常把每一轮比较交换过程称为一次“起泡”。由于每次起泡都使一个最大元素沉到底部，所以比较交换次数除第一轮需进行(n－1)次外，以后可以逐轮少一次。即各轮的比较次数分别为(n－1)，(n－2)，…，2，1 次。

如果排序操作前数列已经按正序排好，则经一轮，即 n－1 次比较，移动次数为零，便可完成排序。最坏的情形是初始序列为逆序的情况，比较次数

$$\sum_{i=1}^{n-1} i = \frac{1}{2}n(n-1)$$

而且每比较一次都要进行一次交换，故移动次数也是最大，为

$3 \cdot \sum_{i=1}^{n-1} i = \frac{3}{2}n(n-1)$（每交换一次，需 3 次数据移动操作）

图 3.25 为对 N 个存储在 A(N)中的数据进行起泡排序的算法。

True BASIC 程序如下：

```
! Bubble sort.
!
dim a(10)
print "Input your data items:"
mat input a(? )
let n=Ubound(a)
for i=1 to n-1
  for j=1 to n-i
    if a(j)>a(j+1) then
      let x=a(j)
      let a(j)=a(j+1)
      let a(j+1)=x
    end if
  next j
next i
mat print a;
end
```

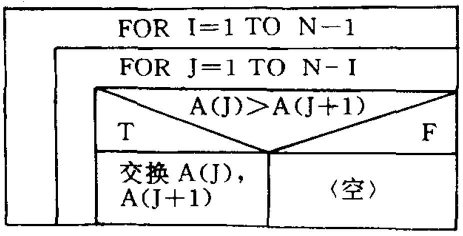


图 3.25

```
Ok. run
Input your data items:
? 7, 5, 3, 9, 1↙
 1  3  5  7  9
Ok.
```

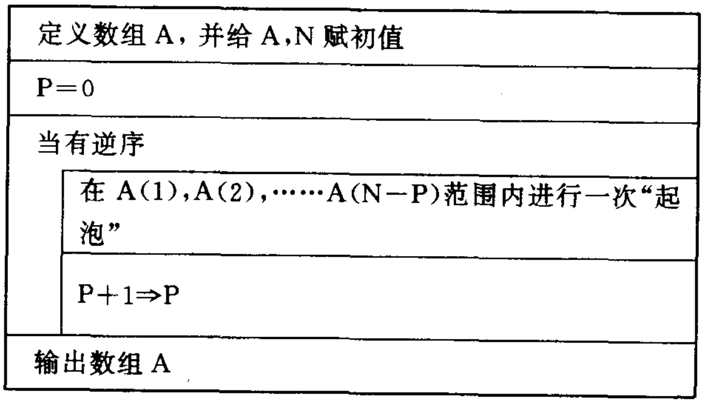


图 3.26

实际上，除非是逆序的情况，一般情况下都不需要进行 n-1 次"起泡"就能排好序。如果当前数据已排好序，则应该结束排序过程。那么计算机如何才能知道当前数据已排好序了呢？一个显而易见的办法是在每一次"起泡"中，记录当前数据中是否存在逆序，如不存在逆序则说明数据已排好序，否则，不能确定是否已排好序，还需进行下一轮"起泡"。由于每进行

一次"起泡"，未排序数据的个数都减 1，如果用变量 P 来记录已进行的"起泡"次数，则算法如图 3.26 所示。

如果用变量 F 来记录当前数据中是否存在逆序(F=1 表示有逆序，F=0 表示无逆序)，则详细算法如图 3.27 所示。

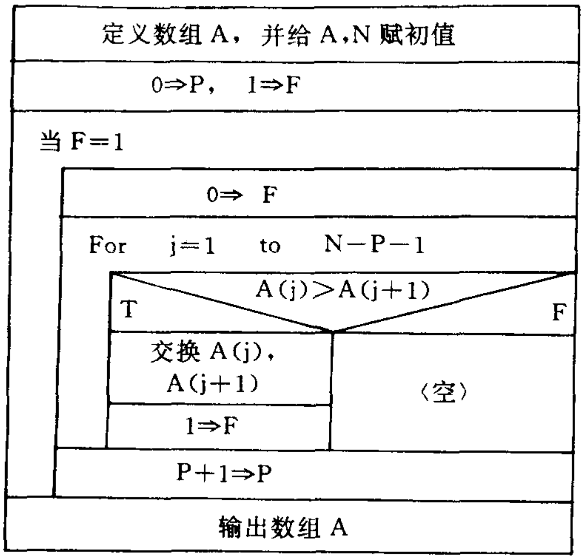


图 3.27

True Basic 程序如下：

```
! Bubble sort
dim A(10)
print "Input your data items:"
mat input A(?)
let N=Ubound(A)
let p=0
let F=1
do while F=1
  let F=0
  for j=1 to N-p-1
    if A(j)>A(j+1) then
      let x=A(j)
      let A(j)=A(j+1)
      let A(j+1)=x
      let F=1
    end if
  next j
  let p=p+1
loop
mat print A;
```

```
end
```

```
Ok. run
Input your data items:
? 7, 5, 3, 9, 1 ↙
  1  3  5  7  9
Ok.
```

习　　题

3.1.1　有一组数,按下列形式存放在 W 数组中

$$W=\begin{bmatrix} 0.00 & 0.12 & 0.13 & 0.14 & 0.15 \\ 1.00 & 1.12 & 1.13 & 1.14 & 1.15 \\ 2.00 & 2.12 & 2.13 & 2.14 & 2.15 \end{bmatrix}$$

若有存储单元 I [0]　K [1]　J [2]

求下列框中“?”所代表的值:

W(I,I)	[?]	W(I+K,J+I)	[?]
(?, ?)	[2.13]	W(I,2)	[?]
W(K,K+J)	[?]	W(?,?)	[1.15]
W(K,J)	[?]	W(J,J)	[?]

3.1.2　用数组存储并在屏幕上打印出四个大字:“欢迎上机”。

3.1.3　下面的 dim 语句定义的数组能存放多少个元素?

dim A$ (1980 to 2000, −10 to 20, 5)

3.1.4　指出下面程序段中的错误

(1)
```
input M,N
dim A(M,N), B$(M)
for I=1 to M
  for J=1 to N
    read A(I,J), B$(I)
  next J
next I
data 1, 2, 3, 4, 5, 6, ……
data A, B, C, D, ……
```

(2)
```
for I=1 to M
  input M,N
  dim A(M,N)
  for J=1 to N
    read A(I,J)
```

```
    next J
  next I
(3) let N=5
  dim A(N)
  for A=1 to 5
    let A(A)=A
  next A
(4) dim A(100)
  mat read A(9)
  for I=1 to 10
    let A(I)=I+A(L)
  next I
  mat print A
  data 9,8,7,6,5,4,3,2,1
  end
```

3.1.5 设计一个模拟洗扑克牌的程序。即将52张牌(除“王”外)随机地排列。

3.1.6 1982年我国有10亿3千万人口。分别以年人口平均增长率为1%,0.5%,0.2%计算,哪一年我国人口就会增长到12亿?用数组存储增长率和年数。

3.1.7 牛的繁殖问题。

有位数学家曾提出这样一道算题:“有一头母牛,它每年年初生一头小母牛。每头小母牛从第四个年头起,每年年初也生一头小母牛。若无牛死亡,问在第二十年时,共有牛多少头?”。

3.1.8 某工厂生产汽车,1月份生产2辆;2月份的产量是1月份的产量减去半辆再翻一番;3月份的产量是二月份的产量减去半辆再翻一番。按此规律,从2月份开始到年底共生产了多少辆汽车?每月各生产了多少辆?

3.1.9 用随机函数产生10个10到30之间的随机整数,利用紧凑格式打印这组数,并指出其中最大的一个数。

3.1.10 某班50人参加一次数学考试,用True BASIC编一程序求全班的平均成绩(用数组存50个学生成绩)。

3.1.11 写一个程序,把下面的数组A改写成数组B

$$A=\begin{bmatrix}11 & 12 & 13 & 14 & 15\\16 & 17 & 18 & 19 & 20\end{bmatrix}\quad B=\begin{bmatrix}20 & 15\\19 & 14\\18 & 13\\17 & 12\\16 & 11\end{bmatrix}$$

3.1.12 某运输公司各类车辆的运费如表3.4所示,各类车辆一周内运行的公里数如表3.5所示。求出各类车辆一周的运费总额,全部车辆的运费总额。

3.1.13 有一个二维数组,请找出其中值最大的元素,以及该元素所在的行和列。

3.1.14 有一个二维数组,求所有靠边的(即第一行和最后一行,第一列和最后一列)元素值之和。

表 3.4

车类＼运费	元/公里
货　车	1.3
拖拉机	0.8
客　车	1.0

表 3.5

车类＼公里数＼星期	一	二	三	四	五	六	七
货　车	1600	2300	4000	3600	7500	2400	3200
拖拉机	7500	6300	7200	4600	5900	6000	1500
客　车	3000	2900	3100	2800	2700	3500	5500

3.1.15 打印杨辉三角形，用数组存储各行数字。

3.1.16 打印一个 7 行 7 列"幻方"，所谓"幻方"，是指一个有奇数行和列的方阵，方阵中的每行每列以及两条对角线上的元素值之和都相等。例如，

11	18	25	2	9
10	12	19	21	3
4	6	13	20	22
23	5	7	14	16
17	24	1	8	15

3.1.17 用高斯消元法解方程组

$$\begin{cases}3x+2y+z=14\\x+y+z=10\\2x+3y-z=1\end{cases}$$

3.2.1 某人设计了一个具有如下功能的程序：输入 n 个学生的姓名及成绩，再打印出来。这是个有错误的程序，请帮他改正。

```
! A program with errors
!
mat read name $ (?), score(?)
mat print name $ ; score,
  data Zhang,93,Li,78,WANG,85,
  data Liu,69,Chen,57,Ma,83
  end
```

3.2.2 某商店帐目应有如下项目：日期、品名、单价、数量、合计。商店每天的销售是随机的，但一般不会超过 120 笔帐。如何设计一个占用内存最小的程序？

3.2.3 设任意两个多项式

$$A = a_0 + a_1x + a_2x^2 + \cdots + a_mx^m$$

$$B = b_0 + b_1x + b_2x^2 + \cdots + b_nx^n$$

请分别设计下面的程序：

(1) A×B；(2) A÷B；(3) 求 A 与 B 的最高公因式。

3.2.4 产生一个随机迷宫：三国时期，诸葛亮困陆逊的八卦阵图，据后人分析，实际上是一个复杂的阵营，使敌人走进后，路径迷失，不容易走出，西方也有类似的建筑。图 3.28 是一个简单的迷宫示意图。

在计算机中，迷宫可以用一个 0 和 1 组成的矩阵来表示，如图 3.28 所示迷宫可以表示为 3.29。

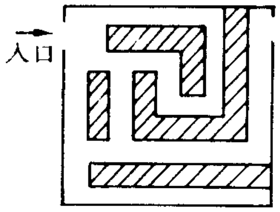


图 3.28

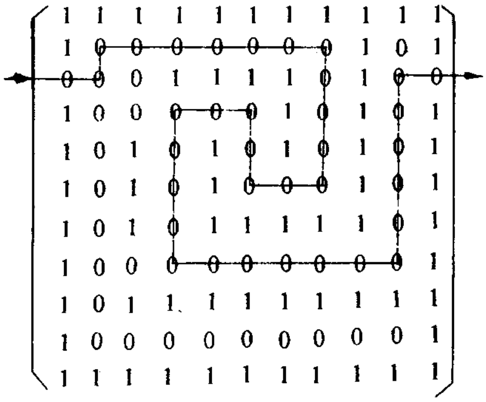


图 3.29

请用随机数产生一个 12×12 的随机迷宫。

请产生一个随机大小的随机迷宫。

3.2.5 形成一个 n×n 的矩阵

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

3.2.6 比高矮：200 个学生任意排成 10 行、10 列的长方形队列。在每一列中选出一个最矮者，然后在所选的 20 个矮人中挑出最高的一人，叫做“矮人中的高个子”。而后这 20 个矮人站回原位去，再在每一行中选一个最高者，又从这 20 名高人中选出最矮的一人，叫做“高人中的矮个子”。请问“矮人中的高个子”与“高人中的矮个子”这二人谁个高？

3.2.7 将一个 8 阶矩阵 A(8,8)按顺时针转 90°。

3.2.8 找出缺少的扑克牌：甲、乙二人玩扑克牌，因对方藏牌，发现少一张牌。请用程序找出这张缺牌来。

3.3.1 某工厂有三个车间，各车间相互提供产品(或劳务)，今年各车间出厂产量及对其它车间的消耗如表 3.6 所示。

表 3.6

消耗系数 车间 \ 车间	1	2	3	出厂产量(万元)	总产量(万元)
1	0.1	0.2	0.45	22	x_1
2	0.2	0.2	0.3	0	x_2
3	0.5	0	0.12	55.6	x_3

表中第一列消耗系数 0.1，0.2，0.5 表示第一车间生产 1 万元的产品分别消耗第一、二、三车间 0.1 万元、0.2 万元、0.5 万元的产品。第二、三列类同。求今年各车间的总产量 x_1，x_2，x_3(用矩阵求解)。

3.3.2 计算

(1) $\begin{pmatrix} 1 & 6 & 4 \\ -4 & 2 & 8 \end{pmatrix} + \begin{pmatrix} -2 & 0 & 1 \\ 2 & -3 & 4 \end{pmatrix}$

(2) $2\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} + 4\begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} + 6\begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix} + 8\begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$

(3) $\begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \\ 3 & 6 & 9 \end{bmatrix} \begin{bmatrix} -1 & -2 & -4 \\ -1 & -2 & -4 \\ 1 & 2 & 4 \end{bmatrix}$

(4) $\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} (1\ 2\ 3)$

3.3.3 四个工厂均能生产甲、乙、丙三种产品，其单位成本如表 3.7 所示。

表 3.7

单位成本 工厂 \ 产品	甲	乙	丙
Ⅰ	3	5	6
Ⅱ	2	4	8
Ⅲ	4	5	5
Ⅳ	4	3	7

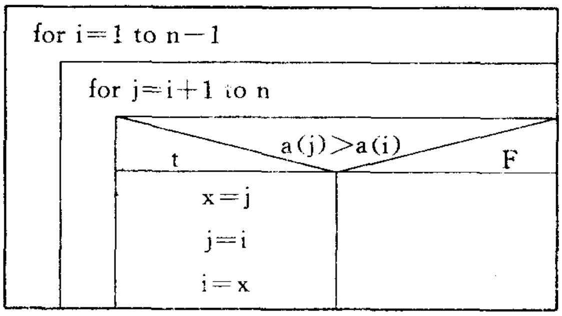


图 3.30

现要生产甲产品 600 件，乙产品 500 件，丙产品 200 件，问由哪个工厂生产成本最低(用矩阵求解)？

3.4.1 有人设计出图 3.30 所示简单选择排序的算法。请分析该程序是否正确？

3.4.2 请给出二分插入排序的算法及程序。

3.4.3 两面插入排序：两面插入排序的基本思想是，另辟一个与原数列大小相同的存

储空间，存储已排序数列，先把第一个数据作为已排序数列的中间元素，再把后面的数据进行二分插入排序，直到把原来的数列都插入到新数列为止。

请给出两面插入排序的算法和程序，并比较两面插入排序与二分插入排序，哪个效率高？

3.4.4 在Shell排序中，当用增量D把原数列分为D个子数列后，对每个子数列可以用插入法进行排序，也可以用起泡法进行排序。请设计用起泡法对各子数列进行排序的Shell排序算法和程序。

3.4.5 摇摆式排序：摇摆式排序是一种改进的起泡排序方法。它与起泡排序的区别在于：第一遍自上而下地比较交换，第二遍则自下而上地进行比较交换，再从上而下，轮番进行，直到把全部数列排完序为止。

请设计摇摆式排序的算法与程序，并将它与起泡法进行比较。

3.4.6 某系举办一次程序设计竞赛，竞赛分四部分进行：

(1) 阅读和理解程序的能力；

(2) 查错和排错的能力；

(3) 算法和数据结构的设计能力。

(4) 程序的风格。

竞赛结果按下列原则排名次：

(1) 先看总分高低；

(2) 总分相同者，看算法和数据结构的设计能力；

(3) 总分以及算法和数据结构的设计能力都相同者，看阅读和理解程序的能力；

(4) 前面的能力都相同者，看查错和排错的能力。

请设计一个按上述原则给参加程序设计竞赛的学生排名次的算法和程序。

3.4.7 设计下面的一维数组操作的算法。

(1) 给定一个数据，将它插入一个已排序的一维数组中。

(2) 给定一个数据，若它已存在一个已排序的一维数组中，则将它删除。

第四章　函数与子程序

4.1　函　　数

4.1.1　自定义函数

在 2.3 节中，已经介绍了一些标准数学函数。这些函数的引用使得程序变得简短而清晰。

例 4.1.1　阶梯问题：爱因斯坦曾出过这样一道题：有一条长长的阶梯。如果你每步跨 2 阶，则最后剩 1 阶；如果你每步跨 3 阶，则最后剩 2 阶；如 果你每步跨 5 阶，则最后剩 4 阶，只有你每步跨 7 阶时，才恰好走完。问这条阶梯最少有多少阶？

根据题意，知道这条阶梯的长度肯定是 7 的整数倍。所以可以从 7 开始，对所有 7 的整数倍的数进行穷举测试，直到找到一个 7 的倍数符合除以 6 余 5、除以 5 余 4、除以 3 余 2、除以 2 余 1 为止。程序如下：

```
!
let n=7
do
   let n=n+7
loop until Mod(n,6)=5 and Mod(n,5)=4 and Mod(n,3)=2 and Mod(n,2)=1
print "Stais of the ladder is:"; n
end
```

也可以将循环部分改为：

```
do
   let n=n+14
loop until Mod(n,6)=5 and(n,5)=4 and Mod(n,3)=2
```

运行结果是相同的：

```
Ok. run
Stais of the ladder is: 119
Ok.
```

如果系统没有提供 MOD 函数，那么每个求模运算都要使用下面的程序段：

```
do
   let r=m-n
   let n=r
loop until r<m
```

```
let mod=r
```

这样不仅给程序设计带来许多麻烦，而且使程序过于庞大，增加了阅读程序的困难和出错的机率。

但是，系统提供的函数是有限的。为了给程序员提供一个提高程序清晰度的环境，多数高级语言都允许用户自己定义一些所需的函数。即用一个或多个语句定义一个函数关系。True BASIC 提供两种不同的格式来定义函数，即：①用一个语句定义一个函数；②用多行语句定义一个函数。

1. 用一个语句定义一个函数（简称单行函数）。单行函数只用一个 DEF 语句定义，其格式为

DEF〈函数名〉（〈参数 1〉，〈参数 2〉，…）=〈表达式〉

例 4.1.2 输入三个数，打印出最大的一个数。程序如下：

```
! Fine the maximum from 3 numbers.
!
def MAX3 (a,b,c)=Max(Max(a,b),c)      ! define a function
print "Enter any 3 numbers:";
input x,y,z
print "The maximum is:"; MAX3(x,y,z)   ! call the function
end
```

```
Ok. run
Enter any 3 numbers:? 5,1,8↙
The maximum is: 8
Ok.
```

上述程序中，用户自己定义了函数 MAX3(a,b,c)，在 def 语句中等号的右边用到了标准函数 MAX。即先用一个 Max 函数在 A 和 B 中找出一个大数，然后再用一个 Max 函数从 Max(a,b)与 c 中找一个大数。这个程序显然比例 2.5.1 中所给出的任何一个程序都要简短、清晰。

2. 用多个语句行定义一个函数（简称多行函数）。有的函数关系是无法用一个语句行定义的，例如有如下函数：

$$y=\begin{cases}0 & x<0\\ 1 & x\geq 0\end{cases}$$

就无法定义为单行函数。True BASIC 允许用多个语句行定义：

```
def y(x)
  let y=0
  if x>=0 then let y=1
end det
```

即：用 DEF 语句开头，以 END DEF 语句结束，中间包含若干个语句。但至少应该有一个 LET 语句，将某一个值赋给所定义的函数（本例中为 y）。其一般格式如下页框内所示。

```
DEF〈函数名〉(〈参数 1〉,〈参数 2〉,…)
…………
LET〈函数名〉=〈表达式〉
…………
END DEF
```

例 4.1.3 计算个人收入调节税:1986 年 9 月 25 日国务院公布了中华人民共和国个人收入调节税条例。条例规定个人所得中的综合收入额按照地区计税基数核算,按月计征。纳税人月综合收入额超过地区计税基数的,就其超基数的三倍以上的部分,按照超倍累进税率征收个人所得税。

如六类和六类以下地区的计税基数为人民币 100 元,则个人综合收入额的 400 元以下部分免税,其它部分如表 4.1 所示。

表 4.1

400 元以上至 500 元部分	20%
500 元以上至 600 元部分	30%
600 元以上至 700 元部分	40%
700 元以上至 800 元部分	50%
800 元以上部分	60%

下面是计算六类和六类以下地区按个人综合收入额计税的程序:

```
! Calculate incometax.
!
det tax(x)
   select case x
      case is<=400
         let tax=0
      case 400 to 500
         let tax=(x-400) *.2
      case 500 to 600
         let tax=(x-500) *.3+20
      case 600 to 700
         let tax=(x-600) *.4+20+30
      case 700 to 800
         let tax=(x-700) *.5+20+30+40
      case is>800
         let tax=(x-800) *.6+20+30+40+50
   end select
end def
print "Input one's income";
input income
print "The income tax is:"; tax (income)
end
```

```
Ok. run
Input one's income? 400↙
The income tax is: 0
Ok. run
```

```
Input one's income? 500↙
The income tax is: 20.
Ok. run
Input one's income? 600↙
The income tax is: 50.
Ok. run
Input one's income? 700↙
The income tax is: 90
Ok. run
Input one's income? 800↙
The income tax is: 140
Ok. run
Input one's income? 1000↙
The income tax is: 260.
Ok.
```

下面对自定义函数再作几点说明：

(1) 函数名：函数名是一个函数区别于其它函数的标识。函数名隐含说明了函数的类型。数值函数名遵循数值变量的命名规则，字符串函数名遵循字符串变量的命名规则，即它以“$”符号结尾。

为提高程序的可读性，函数名也应按“见名知义”的原则命名。

系统提供的一些无参标准函数名，它们的名字不能用作 True BASIC 自定义函数名。它们是：DATE，DATE $ ，DET，EXLINE $ ，EXTEXT $ ，EXTYPE，MAXNUM，PI，RND，TIME，TIME $ 。

此外，在同一程序中，用户所定义的函数名，不能再用作变量名或数组名。如下面的程序是错误的：

```
⋮
def a(m)
  let a=2∧m
end def
⋮
let a=2
let b=2*a∧4
```

(2) 函数类型：每个函数将返回一个值。这个值可以是数值型的，也可以是字符串型的。返回数值数据的函数称为数值函数，返回字符串数据的称为字符串函数。

例 4.1.4 给定一个字符，要求输出几个同样的字符。

```
!
!
def repeat $ (a $ , n)
  let rep $ =""                       ! start with anull string
for i=1 to n                          ! take chars in reverse
```

```
      let rep$ =rep$ &A$                    ! add
    next i
    let repeat$ =rep$
  end def
  input prompt "Enter a character": x$
  input prompt "How times repeated?": m
  print repeat$ (X$ ,m)
  end
```

```
Ok. run
Enter a character x↙
How times repeated? 5↙
xxxxx
Ok. run
Enter a character: o↙
How times repeated? 10↙
0000000000
Ok.
```

(3) 函数的调用：在定义了一个函数之后，可以在同一个程序中多次调用。例如，定义一个函数：

def Area(r)=pi * r∧2

在程序中调用此函数的方法与调用标准函数的方法完全相同。唯一不同的是：自定义函数需要用户先定义一个函数才能使用该函数，而标准函数是系统已定义好的，不必用户自己定义。

程序中可以用以下语句打印当半径 r=1,2,3,4 时圆的面积：

Print Area(1), Area(2), Area(3), Area(4)

一般将定义函数中的参数(parameter)称为形式参数或虚拟参数(或虚变量、虚元)。例如上面 Area 函数的 r 就是一个形式参数。它并不是实在存在的变量，它并不占内存单元。只有在调用函数时，才临时开辟一个内存单元，存放 r 实际的值。在调用函数时，所用的自变量(argument)一般称之为实在参数(或实变量、实元)。例如，Area(1)，Area(2)中的数"1"和"2"就是实际参数。

在调用函数时，将实在参数对应地取代形式参数，就可得到函数的值，例如：

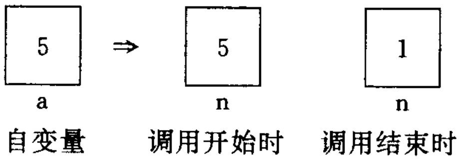
```
def c(a,b)=sqr(a * a+b * b)
   ⋮
let s=c(3,4)
```

即，将 3 和 4 分别代替函数的虚参 a 和 b，从 def 语句中的等号右边的表达式计算 sqr(3 * 3 +4 * 4)，得到值"5"，也就是，c(3,4)和 s 的值为 5。

在调用函数时，计算机系统是这样执行的：先将实际参数的值传送到一个临时存储单元中。再将此临时变量的值拿来进行运算。因此，若在调用函数过程中形式参数的值发生变化，也不会影响实在变量原来的值。例如，

```
def nfactor (n)
  let factor=1
  do
    let factor=factor  * n
    let n=n-1
  loop until n<=1
  let nfactor=factor
end def
  ⋮
    let a=5
    let t=nfactor(a)
      ⋮
```

在调用时，实在参数 a 的值为 5，在调用过程中，n 的值不断变化，到调用结束时，n 的值为 1。但由于 a 和 n 是两个不同的存储单元，因此，n 值的变化不影响 a 的值，在调用结束后，a 的值仍为 5。



定义函数时，形式参数名并不重要，可以任意换用另一个名字，例如，

```
def Area(r)=pi * r ∧ 2
```

也可以改写为

```
def Area (radio)=pi * radio ∧ 2
```

或

```
def Area(x)=pi * x ∧ 2
```

在调用时，方法是完全一样的，即

```
print Area(1), Area(2),Area(3),Area(4)
```

也就是说，定义函数时，只是定义一个函数关系，函数名和参数名都不是最本质的，是可以改变的，这如同在代数中一样，例如，

$$y(x)=x^2+x+1$$

$$y(u)=u^2+u+1$$

或

$$f(x)=x^2+x+1$$

都是代表同一个函数关系，如果在坐标纸上画出曲线，曲线形状是相同的，只是自变量和应变量的名字改变而已。

还应该说明，定义一个函数时，实际上并没有进行任何操作，只不过把定义的这个函数关系记下来，只有在调用时，才进行运算并得到结果。

形式参数与实在参数的关系，可以用以下比方来说明：莎士比亚塑造了哈姆雷特这个角色，但真正出现在舞台上的扮演哈姆雷特的人可能是张三，也可能是李四，而且张三和李四还可以在其它的剧中扮演其它的角色。虚参就好比是角色，而实参就好比是演员。这种数据传送的方式称为虚实结合。

在调用时，将实在参数的值传送到一个临时存储单元中。

(4) 函数可以有参数，也可以没有参数。例如，例 4.1.4 中函数 REPEAT 的参数是 A$,N。在程序中调用此函数时，应该在括号内相应地给出实在参数。调用带参函数时，必须给出参数。如例 4.1.4 不可以写为

```
⋮
def repeat $ (a $ ,n)
  let repeat $ ="  "
  for i=1 to n
★ let repeat $ =repeat $ &a $
  next i
end def
⋮
```

上面程序段中第四行(带有★标记的行)有错。在定义多行函数时,函数名(不带参数)只能出现在赋值号的左边,作为给该函数赋值。第四行中赋值号右边出现不带参数的函数名 REPEAT $,这是不允许的。如果是调用函数,必须带参数。

字符串或数值型常数、变量和数组都可以用来作参数。在定义函数时,如果用数组作为形式参数,要用括号和逗号来表示数组的维数,例如用 a(,),表示 a 是一个二维数组参数,而 A( )是一个一维数组参数。但在调用函数时,括号内只写数组名而不必用逗号表示维数。另外,做为形式参数的数组名不必用 dim 语句说明。

例 4.1.5 用数组作参数和自变量。

```
! Using arrays as parameters and arguments.
!
dim B(10,10)
def First (A(,))=A(2,1) * 2           ! parameter must have(,)
input B(2,1)
print First (B)                       ! argument must not have(,)
end
```

```
Ok. run
? 5↙
 10
Ok.
```

定义一个自定函数 First,用一个二维数组 A 作形式参数,其值是其中的数组元素 A(2,1)乘 2。程序接收 5,送给数组元素 B(2,1),用 print 语句调用函数 First 时,将 B(2,1)的值 5 和 B 数组中其它元素的值(今为 0)传送给子程序中的数组参数 A,将 A(2,1)再乘以 2 得 10。从而使 First(B)的值为 10。这里 B 是实在数组,A 是虚拟数组,执行的情况如同 First(B)=B(2,1) * 2,即将实在参数代入虚拟参数。

传送数组给函数时必须谨慎。因为函数被调用一次,都要将整个数组传送到一个临时数组中,会占用许多内存单元,影响程序的效率。

此外,要注意在函数中不可用通道作参数。

(5) 参数名只在函数范围内有效,或者说它的作用是局部的。如果在函数定义范围之外有一个同名的变量,它们之间没有任何关系。

例 4.1.6 用函数计算 C_m^n,即 $\frac{m!}{n! \times (m-n)!}$

```
! Using a function to calculating value
! of combination size N in size M.
def Fact(N)
  if n=0 or n=1 then
    let Fact=1
    exit def
  else
    let f=1
    for i=1 to n
      let f=f * i
    next i
    let Fact=f
  end if
end def
print "Enter two positive integer: N & M";
input n,m
let Comb=Fact(m)/(Fact(n) * Fact(m-N))
print "The combination value of size N";
print "in size M is:"; Comb
end
```

```
Ok. run
Enter two positive integer: N & M? 2,3↙
The combination value of size N in size M is: 3
Ok. run
Enter two positive integer: N & M? 3,4↙
The combination value of size N in size M is: 4
Ok.
```

上述程序中，函数定义中的参数 N 与程序中的变量 N 是两个不同的量，它们是各自独立的。

(6) 在例 4.1.6 中，使用了 EXIT DEF 语句。EXIT DEF 语句可以出现在函数定义中的任何地方。执行此语句时，函数就返回到其调用处，形成函数的非终端出口。

(7) 函数必须定义后再引用，如果想先引用后用 def 定义，则必须先用 DECLARE DEF 语句说明。DECLARE DEF 语句的作用是声明："后面将定义一个(或若干个)函数。"DECLARE DEF 语句的格式为

DECLARE DEF〈函数名 1〉,〈函数名 2〉,……

如果事先不用 DEF 语句定义函数，又没有用 DECLARE DEF 语句声明，将会使系统分不清被引用的到底是函数还是数组。请看例 4.1.2 写成下面的形式后运行产生的结果。

```
! Fine the maximum from 3 numbers.
!
print "Enter any 3 numbers";
```

```
  input x,y,z
★print "The maximum is:"; MAX3(x,y,z)          ! call the function
  def Max3(a,b,c)=Max(Max(a,b),c)              ! define a function
  end
```

```
Ok. run
This must first appear in a DIM or DECLARE DEF.
```

这时光标将停在带“★”标记的程序行中的“M”处，要求先用 DIM 或 DECLARE DEF 语句加以说明。请看下面的程序及运行结果。

```
  ! Fine the maximum from 3 numbers.
  !
★declare def Max3
  print "Enter any 3 numbers";
  input x,y,z
  print "The maximum is:"; Max3(x,y,z)          ! call the function
  def Max3(a,b,c)=Max(Max(a,b),c)               ! define a function
  end
```

```
Ok. run
Enter any 3 numbers? 3,5,7↙
The maximum is: 7
Ok.
```

(8) 关键字“DEF”可以用“FUNCTION”代替。如前面的程序可改为

```
  ! Fine the maximum from 3 numbers.
  !
  declare def Max3
  print "Enter any 3 numbers:";
  input x,y,z
  print "The maximum is:"; Max3(x,y,z)          ! call the function
★function Max3(a,b,c)=Max(Max(a,b),c)          ! define a function
  end
```

```
Ok. run
Enter any 3 numbers: ? 3,1,5↙
The maximum is: 5
Ok.
```

4.1.2 外部函数——使用局部变量

函数定义在其调用程序内，称为内部函数。函数定义在其调用程序外部，称为外部函数。在 4.1.1 节中所举的例子都是内部函数。我们已经讲过，参数名的有效范围只限于函数内

部，即参数名是“局部”的。但是函数中除了使用参数外，还要使用一些变量。这些变量既在函数范围内有效，也在程序中函数之外的范围有效。例如，

```
def y(x)
  if x>0 then let y=10
  else if x=0 then let y=0
  else let y=-10
  end if
  let y=y/n
end def
  ⋮
let n=5
print y(3)
  ⋮
```

变量 n 在函数定义范围内和函数定义范围外都出现，由于它并不是函数的参数，因此它不是局部变量。n 在函数定义范围内和程序中其它场合都是有效的，它是同一个变量。由于程序中赋给 n 的值为 5，因此，调用函数 y 后，得到 y(3)的值为 2。

这个 n 就是一个“全局变量”，或“全程变量”(global variable)。在使用全程变量时应十分注意。如果考虑不周，会引起程序中不同部分的互相干扰。请看下面的例子。

例 4.1.7 一个隐藏有错误的例子。

```
! An insidious bug.
!
det repeat $ (a $ ,n)
  let rep $ =""                  ! start with anull string
  for i=1 to n                   ! take chars in reverse
    let rep $ =rep $ & A $       ! add
  next i
  let repeat $ =rep $
end det
input prompt "Enter a character:":x $
input prompt "How times repeated?": m
for i=1 to m
  print repeat $ (x $ ,m)
next i
end
```

```
Ok. run
Enter a character: X↙
How times repeated? 10↙
XXXXXXXXXX
Ok.
```

该程序要求打印一个字母方阵(每调用一次 repeat $,应打印出一行“X”)，结果仅打了

一列，原因在何处呢？原因就在于在函数中使用了一个循环变量 i，在程序中也使用了一个循环变量 i。执行程序，进入 FOR 循环后，开始 i＝1；调用一次函数 repeat $ (x $,m)后，i 变为 m＋1。由于 i 是全程的，返回主程序后，因循环变量已超过循环终值，于是只打了一行字符便退出主程序中的循环。

使用全程变量容易使程序隐藏错误。当程序规模较大，把程序分交给多个人去分别编写时，可能会出现互相干扰的情况。稍有不慎便会埋下祸根。如果好几个人都使用同一个变量，就会产生混乱。解决这一问题的一种措施是尽量不使用同名变量。但在程序中用到的变量数很多时，特别是在不同的人编写一个程序的不同部分时，这一点往往是难以做到的。最根本的办法是使变量与参数一样都局部化。这种变量称为局部变量(Local variable)。外部函数便可以做到这一点。

外部函数被定义在主程序之外，即在主程序的 END 语句之后。它作为一个独立的模块出现。下面是将例 4.1.7 中的函数写成外部函数后，程序及其运行情形。

```
declare def repeat $
input prompt "Enter a character:"; x $
input prompt "How times repeated?": m
for i=1 to m
  print repeat $ (x $ ,m)
next i
end
def repeat $ (A $ ,n)
  let rep $ =""                    ! start with anull string
  for i=1 to n                     ! take chars in reverse
    let rep $ =rep $ & A $         ! add
  next i
  let repeat $ =rep $
end def
```

```
Ok. run
Enter a character: X↙
How times repeated? 10↙
XXXXXXXXXX
XXXXXXXXXX
XXXXXXXXXX
XXXXXXXXXX
XXXXXXXXXX
XXXXXXXXXX
XXXXXXXXXX
XXXXXXXXXX
XXXXXXXXXX
XXXXXXXXXX
Ok.
```

显然，程序得到了预期的结果。

由于外部函数定义在其调用程序之外，因此在引用该外部函数之前必须先用 DECLARE 语句声明。

上面的程序中，包含了两个程序单位：主程序和外部函数，或者说，有两个模块。而内部函数和主程序是属于同一个程序单位，它们是在一个模块之中。

内部函数与其 调用程序之间不仅通过参数传递数据，还要通过同名变量传递数据。而外部函数与其调用程序之间仅通过参数传递数据，即使它们之间有的变量是同名的，但互相没有关系，被看作是不同的变量，好象南京的新街口与北京的新街口不是一个地方一样。或者说当我们每调用一次外部函数时，它的所有的变量都被重新初始化一次。

通道对外部函数也是局部的。如果一个外部函数打开了某通道(如打印机通道)，则当退出这个函数时，该通道也将被关闭。

例 4.1.8 比较下面两个程序口：

程序 1 及其运行结果：

```
! Demonstrate channel parameters
!
def a$ (m)
  open #90: printer
  let x$ =" "
  for i=1 to m
    let x$ =x$ & "X"
  next i
  let a$ =x$
  print #90: "Sub......";x$
end det
let n=10
print "Main......";a$(n)
print #90: "Main * * * * * * * * * *"
close #90
end
```

```
Ok. run
Main......XXXXXXXXXX
Ok.
```

同时在打印机上打印出

```
Sub......XXXXXXXXXX
Main * * * * * * * * * *
```

程序 2 及其运行结果：

```
! Demonstrate channel parameters.
!
```

```
  declare def a $
  let n=10
  print "Main......"; a $ (n)
★print #90: "Main * * * * * * * * * * *"
  close #90
  end
  def a $ (m)
    open #90: printer
    let x $ =" "
    for i=1 to m
      let x $ =x $ & "x"
    next i
    let a $ =x $
    print #90: "Sub......"; x $
  end def
```

```
Ok. run
Main......XXXXXXXXXX
Channel isn't open. (光标停在有"★"标记的程序行)
```

在打印机上只打印出：

```
Sub......XXXXXXXXXX
```

这是因为程序 2 中打开打印机通道的操作是在外部子程序中进行的。而程序 1 中打开打印机通道的操作是在内部子程序中进行的。前者的通道是局部的，后者是全局的。

4.1.3 递归函数

有时，在函数的定义中要用到另外一些函数。如把求 C_m^n 的函数定义为一个 COMB(M，N)，它利用求阶乘的函数 Fact。

```
! Using a recursive subroutine to calculating value
! of combination size N in size M.
def Fact(n)
  if n=0 or n=1 then
    let Fact=1
    exit def
  else
    let f=1
    for i=1 to n
      let i=i * f
    next j
    let Fact=f
  end if
end def
```

```
det Comb(m,n)
  let Comb=Fact(m)/(Fact(n) * Fact(m-n))
end def
print "Enter two positive integer: N & M";
input a,b
print "The combination value of size N";
print "in size M is:"; Comb(b,a)
end
```

```
Ok. run
Enter two positive integer: N & M? 2,3↙
The combination value of size N in size M is:3
Ok. run
Enter two positive integer: N & M? 3,4↙
The combination value of size N in size M is: 4
Ok.
```

在函数 Comb 的定义中使用了函数 Fact。这种情形叫做函数的嵌套。

True BASIC 还允许一个函数部分地由它自己组成，或由它自己定义。这种函数称为递归函数(recursion)。例如，以前曾介绍过可以用迭代的方法定义求阶乘的函数。它是基于下面的递推公式的：

$$\begin{cases} FACT(n)=1 & (n=0,1) \\ FACT(n)=FACT(n-1)*n & (n>1) \end{cases}$$

即从 FACT(n－1)可以推出 FACT(n)。也可以把这一定义看成是"递归"的，即求函数 FACT(n)，又要用到 FACT 函数。或者说，函数由它自己定义的。按递归定义写出的递归函数如下：

```
! Recurtive factorial.
!
def Fact(n)
  if n<=1 then let Fact=1 else let Fact=n * Fact(n-1)
end def
input m
print Fact(m)
end
```

```
Ok. run
? 5↙
  120
Ok. run
? 3↙
  6
Ok.
```

递归是一种有力的武器。有可能用有限的语句来定义对象的无限集合。例如，当你背靠一个大衣镜再在面前放一个镜子时，镜子中便出现了这样一种景象：你照镜子，镜子中又有一个你在照镜子，……。这一现象中的对象是无穷的，但是使用递归这一武器便可以用有限的语句定义它：某人在照镜子，照到的是“他对着镜子在照镜子”。

有些递归过程不是无穷的，它存在一个边界。如求阶乘的运算就是一个有穷递归过程：阶乘函数调用阶乘函数，直到 N<=1 为止。N<=1 时 FACT=1 就称为递归地求阶乘的“边界条件”。下面的例子最能清楚地说明递归的概念：有五个人坐在一排，问第五个人的岁数。他回答：比第四个人的岁数大两岁。为了知道第五个人的年龄就必须先知道第四个人的年龄。于是去问第四个人的年龄。回答是：比第三个人的岁数大两岁。为了知道第四人的年龄必须先知道第三个人的年龄。而第三个人又比第二个人大两岁，第二人又比第一个人大两岁。最后问第一个的年龄，回答是 15 岁。今问第五个人的年龄。

为了求第五个人的年龄，必须“调用”第四个人的年龄，即 AGE(5)=AGE(4)+2。而 AGE(4)的值仍然不知道，又要调用第三个人的年龄，AGE(4)=AGE(3)+2。如此调用下去，直到找到 AGE(1)=15 为止，并由 AGE(1)开始求 AGE(2)=AGE(1)+2=15+2=17，AGE(3)=AGE(2)+2=19，AGE(4)=AGE(3)+2=19+2=21，AGE(5)=AGE(4)+2=21+2=23。这时才求出第五个人的年龄。这一过程可以递归地定义为

$$\begin{cases} AGE(N)=AGE(N-1)+2 & (N>1) \\ AGE(N)=15 & (N=1) \end{cases}$$

N=1 时，AGE(N)=15，这便是求年龄函数的“边界条件”。

应当注意递归与递推的区别。递推是从已知的初始条件出发，逐次求出最后需要求的值。而递归则从需要求的未知项出发，逐次上溯调用其本身的求解过程，直到初始(边界)条件。递归与迭代也不同。迭代的含义是重复，而递归则是自己调用自己。每一个递归过程都可划分为递归调用与返回两个子过程。下面以用递归法求契波那西(Fibonacci)序列为例来说明这一过程。

用函数形式表示 Fibonacci 数列中的第 n 个数：

$$FIB(n)=\begin{cases} 1 & (n=1) \\ 1 & (n=2) \\ FIB(n-1)+FIB(n-2) & (n>2) \end{cases}$$

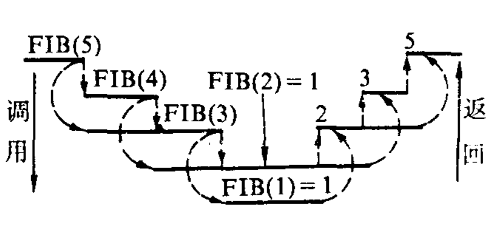


图 4.1 FIB(5)的递归求解过程

如果想求第五个数，可以用函数 FIB(5)。由于 n=5，故 FIB(5)=FIB(4)+FIB(3)，而 FIB(4)=FIB(3)+FIB(2)，FIB(3)=FIB(2)+FIB(1)，FIB(2)和 FIB(1)都等于 1。其递归过程见图 4.1。

当进入递归过程后便逐层向下调用 FIB(n)函数，直到到达初始条件(FIB(1)，FIB(2)都等于 1)，也即递归的终止条件为止。接着进入返回过程，由递归终止条件所给定的特定解，按照原来的路径返回，得到各层上相应的函数值，即 FIB(3)=2，FIB(4)=3，FIB(5)=5。

递归是一种十分有用的程序设计技术。递归函数的应用使得函数的定义和算法描述比使用非递归方法简洁易读、可理解性好。许多问题用递归过程来描述它们不仅非常自然，而

且正确性证明也比相应的非递归形式容易得多。

必须指出的是，由于外部函数中所有的变量对于函数都是局部的。在每次递归调用时，函数的全部变量都全部重新赋值。因而，外部函数可以正确地实现递归调用。用内部函数实行递归调用时则要谨慎，因为可能会有全局变量的干扰。

4.1.4 按功能定义函数

前面介绍了函数的定义与引用方面的基本知识。学习了这些内容后，读者便会想到：当程序中需要重复引用一个复杂表达式进行计算时，便可以用一个单行函数去代表；当程序中需要引用一个产生单值的程序段时，便可以用一个多行函数去代表它。这样会使程序更简练、清晰。其实问题并非如此简单，因为这样定义的函数是没有任何规律的，只是使用上方便了一些，而每个函数不是按它的功能来区分，也就是说它在概念上还是不明确的。随着程序中定义的功能不明确的函数的增加，程序的可理解性会变得越来越差。好像一届运动会上把游泳、乒乓球、100 米和铅球编为一组，而把标枪、拳击、举重、自行车和滑冰编为另一组，……一样，这样编的组越多，人们越觉得难以理出头绪。

那么，怎样定义函数才是概念明确的呢？经过多年的实践，人们总结出这样一条经验：按功能划分出的函数在概念上是明确的。也就是说，每个函数应当具有一个完整的功能。当程序中包括的功能较多时，对于每一个产生单值结果的功能，都可以把它们定义成函数。这样一个大的程序就有可能分解成许多小模块。尤其是外部函数只通过参数接收调用程序传递的数据，并最后向主程序返回一个单值，只要搞清各个函数的功能及其与调用程序间的"接口"关系，就可以把各个函数分给不同的人分头编写、调试以及编译。从这个意义上说，函数是分解程序的一种手段，分解的准则是要求每个函数有单一而独立的功能。

在本节中定义的函数都是按功能定义的。如在例 4.1.2 中把"从三个数中找一个最大数"定义为一个函数。当给出问题的抽象算法时，便得到了主程序的算法，如图 4.2 所示。对抽象算法的细化，不是把主程序加细，而是用函数，即另外的程序段去描述。这种程序设计方法称为自顶(高层次的抽象)向下(具体)、逐步细化的程序设计。当程序比较复杂时，要采用把程序分成几个程序单位(模块)、一个外部函数(或下面要介绍的子程序)是一个模块。这就是模块化的设计方法。在结构化程序设计中，要充分利用模块化方法。在本章 4.4 节中将进一步讨论模块化的问题。

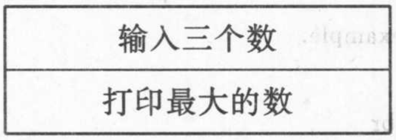


图 4.2

4.2 子 程 序

子程序(Subroutine 或 Subprogram)是 True BASIC 提供的把程序分解为较小单位的另一种手段。它与函数有以下几点显著的区别：

1. 一个函数仅返回一个值到调用程序中，而子程序可以返回多个值。

2. 函数调用在表达式中的地位象引用一个变量一样，而子程序要用一个专门的语句去调用。

3. 定义在调用语句后的子程序不需用 DECLARE DEF 语句声明。

下面主要介绍子程序的定义与调用方式。读者将会了解到子程序与函数的异同。

4.2.1 子程序的定义和调用

当一个程序中包含多个功能时，每个功能都可以用一个子程序分别去实现。每个子程序有一个名字，在主程序中用 CALL 语句调用。CALL 语句的格式为

```
CALL    〈子程序名〉
```

子程序名的命名规则与数值变量相同，即由字母开始，最大可达 31 个字符的字母(不区分大小写)，数字和下划线"—"的序列表示。子程序名不能以"＄"结尾。Divide，pack，poke 是 True BASIC 的保留字，不能用作内部子程序名。

一个名字用于标识子程序后，便不可再用于标识变量和函数了。

定义子程序以 SUB 开头，以 END SUB 结束，其格式为

```
SUB〈子程序名〉
    ………
    ………  }〈语句块〉
    ………
END SUB
```

当调用程序执行 CALL 语句时，控制就会转向 CALL 语句指定的子程序，执行子程序中的各个语句，直到遇到一个 EXIT SUB 语句或 END SUB 语句，然后返回到主程序中调用它的 CALL 语句的下一语句。这一调用过程好象把子程序插入到调用处一样。

例 4.2.1

```
! An example.
!
sub subr
  print "*";
end sub
for i=1 to 10
  call subr
next i
end
```

```
Ok. run
**********
Ok.
```

在上面的程序中，把"打印"作为一个独立功能定义为一个子程序，主程序可以重复地调用它。每调用一次打印出一个"*"。

4.2.2 内部子程序与外部子程序

与函数一样，子程序可以定义在 END 语句之前(如例 4.2.1)与其调用程序一起组成一

个程序单位；也可以定义在 END 语句之后，形成一个独立的程序单位。前者称为内部子程序，后者称为外部子程序。内部子程序使用的变量(指非参数的其它变量)是全程变量，而外部子程序使用的变量是局部变量。

例 4.2.2 比较下面两个程序及执行结果：

程序 1

```
! Program 1
!
sub A
  let X=2*X
  print X,
  let Y=2
end sub
let X=5
call A
print Y
end
--------------------
Ok. run
  10        2
Ok.
```

程序 2

```
! Program 2
!
let X=5
call A
print Y
end
sub A
  let X=2*X
  print X,
  let Y=2
end sub
--------------------
Ok. run
0        0
Ok.
```

程序 1 使用了内部子程序，主程序与子程序中的每一同名变量都是同一变量。所以主程序先给 X 赋值 5 后再调用子程序 A，A 中的 X 值便为 5，再经赋值 LET X=2*X，显示出“10”。然后在子程序中给 y 赋值 2，又传给主程序显示出“2”。

程序 2 使用了外部子程序。主程序与子程序形成两个独立的程序单位。这两个程序单位中虽有同名变量，却好象是两个家庭中的两个同名孩子一样，代表了两个不同的变量。所以，子程序中的 x 与主程序中的 y 都因没有赋过初值而被当作“0”显示出来。

True BASIC 中，每程序单位不仅有自己的局部变量，而且数据区与数据指针也是局部的。每个程序单位内部的 READ 语句只能从本程序单位的数据区中读取数据，并且每个程序单位内部的 RESTORE 语句只能恢复自己的数据区的指针。

例 4.2.3 比较下面的两个程序

程序 1

```
! Program 1
!
sub D
  for i=1 to 3
    read A
    restore
    print A;
  next i
  data 5
```

程序 2

```
! Program 2
!
data 2
for i=1 to3
  read B
  restore
  print B;
next i
call D
```

```
end sub
data 2
for i=1 to 3
  read B
restore
print B;
next i
call D
end
```

```
end
sub D
  for i=1 to 3
    read A
    restore
    print A;
  next i
  data 5
end sub
```

```
Ok. run
5 5 5 5 5 5
Ok.
```

```
Ok. run
2 2 2 5 5 5
Ok.
```

程序 1 使用了内部子程序，整个程序是一个程序单位，两个 DATA 语句建立了一个数据区"5，2"。在主程序与子程序中 B 与 A 分别读三次数据，但每读一次便恢复一次数据指针，所以读的总是"5"，显示出六个"5"。程序 2 使用了外部子程序，A，B 分别从各自的数据区中读数据，所以显示出三个 2(B 的值)和三个 5(A 的值)。

4.2.3 带参子程序——虚实结合

前面列举的子程序都是不带参数的。它既不接收主程序传送的数据，也不向主程序传送数据，只是在主程序调用它时，执行指定的操作。主程序与子程序之间的这种联系，称为非直接耦合，是一种理想的耦合关系。这一点与无参函数不同。无参函数(如 PI，DATA，TIME，RND，……)虽不要主程序向它传送数据，但却要向主程序返回一个数据。

```
SUB<子程序名>(<参数 1>,<参数 2>,…)
   .........
   .........  }<语句块>
   .........
END SUB
```

True BASIC 还允许使用带参子程序。每个子程序允许带有多个参数。这些参数可分为以下两种类型：

(1) 入口参数——用于接受主程序传送的数据，在子程序中使用，其值不被改变。

(2) 出口参数——用于向主程序传送数据，其值可以被子程序中的语句改变。这一点也与函数不同。函数的参数只能用来接收调用程序传送的数据，没有输出功能，输出是由函数名(由函数定义中的 LET 语句赋值)进行的。

带参子程序可以用下面的格式定义：

调用程序需要子程序所规定的功能时，可以使用下面的调用语句格式：

```
CALL <子程序名> (<参数 1>,<参数 2>,……)
```

与函数一样，子程序中的参数称为形式参数(或虚参数)，调用语句中使用的参数称为实

在参数(或实参数)。实参和形参在个数、顺序和类型上都应一一对应。当子程序被调用时,入口参数将由对应的实参替换形参进入子程序中参加操作,并在出口处把出口参数的值传送给调用程序中对应的实参。通过参数传递形成子程序与调用程序间的一个数据接口。当使用外部子程序时,只有通过参数传递才能与外界发生数据联系。也就是说,这种数据接口是唯一的。外部子程序与调用程序间的这种联系称为数据耦合。数据耦合是仅次于非直接耦合的一种较好的耦合关系。当使用内部子程序时,数据接口便不是唯一的。除参数之间的数据传送外,还可以通过全程变量传送数据。这种耦合关系称为公共耦合,由于有不止一个途径传送数据,容易引起差错,因此是一种较差的耦合关系。

例 4.2.4 用子程序计算 C_m^n:在例 4.1.6 中使用了函数来计算 C_m^n。这里用子程序来处理。

首先,把问题分解为三个子问题:

(1) 输入两个数 M,N,用子程序 IN(M,N)实现这一功能。其中 M,N 是子程序 IN 的出口参数。

(2) 计算 C_m^n,用子程序 COMB(C,M,N)来实现这一功能。其中 C 是计算结果,是子程序 COMB 的出口参数,M,N 是子程序 COMB 的入口参数。

(3) 输出结果 C,用子程序 OUT(C)来实现这一功能,其中 C 是子程序 OUT 的入口参数。

于是可以得到如下主程序:

```
! Using subroutines to compute value
! of combination of size N in size M.
!
call IN(M,N)
call COMB (C,M,N)
call OUT(C)
end
```

主程序从全局出发描述了系统的功能。这些功能用三个子程序分别完成。这是“顶层设计”,即最高一层的分解。下面再考虑如何实现这三个子程序。

(1) 子程序 IN(M,N)的功能是输入两个数 M,N。程序很简单,可以直接写出如下:

```
sub IN(M,N)
  print "Enter two positive integer M,N"
  input M,N
end sub
```

(2) 子程序 OUT(C)就是输出 C 代表的 C_m^n 的值,也可以直接写出:

```
sub OUT(C)
  print "The value of combination of";
  print "size N in size M is:"; C
end sub
```

(3) 子程序 COMB 是考虑的重点。它的功能是接收主程序传送来的参数 M,N,传送给

主程序参数 C。如图 4.3 所示，在主程序中，把函数 COMB 看作一个“黑箱”，它输入 M 与 N，输出 C。我们的任务是打开“黑箱”去构造它的内部。

图 4.3

由数学公式

$$C_m^n=\frac{m!}{n!\ (m-n)!}=m!\ /n!\ /(m-n)!$$

可知，求 C_m^n 可以由阶乘运算和除运算两种功能实现。即先求出 F1＝m!，F2＝n!，然后求 D＝F1/F2，再求 F3＝(M－N)!，最后再用 D 除以 F3，求出 C＝D/F3。因此，可以把子程序 COMB(C，M，N)写为

```
sub COMB(C,M,N)
  call FACT (F1,M)            (求 F1=M!)
  call FACT (F2,N)            (求 F2=N!)
  call DIV (D,F1,F2)          (求 D=F1/F2)
  call FACT(F3,(M-N))         (求 F3=(M-N)!)
  call DIV(C,D,F3)            (求 C=D/F3)
end sub
```

子程序 COMB 比图 4.3 较详细地描述了求 C_m^n 的过程。或者说，它把求 C_m^n 的过程分解为另外两个子程序：FACT 和 DIV。

FACT 是求阶乘，由以前的讨论，可写出如下子程序：

```
sub FACT (FAC, NUM)
  let FAC=1
  if NUM<=1 then
    exit sub
  else
    for I=1 to NUM
      let FAC=FAC * I
    next I
  end if
end sub
```

DIV 是求商。也很容易写出：

```
sub DIV(QUOT, N1,N2)
  let QUOT=N1/N2
end sub
```

到此为止，再没有需要进一步分解的功能了。程序也就设计好了。按照程序单位之间的调用关系，可以画出图 4.4 所示的程序结构。这种结构呈层次状，或称树状(倒挂的树)。每一个方框称为一个程序模块(programming module)。设计程序的过程是，先设计最上层的主模块(主程序)，然后把它用下一层的几个模块去描述。按此方法一直向下，直到程序被描述得足够详细为止。这一过程好象一棵树的生长过程一样，先生成树根(主模块)，然后向上

逐层分枝，直到每个树叶。不过这里的树是倒挂的。这种方法称为自顶向下的逐步细化方法。关于这一方法，将在4.4节中进一步介绍。

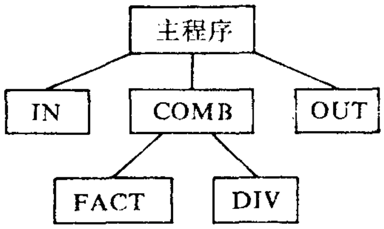


图 4.4

使用子程序时，数据传递的实现方法与使用函数时不同。对函数，参数是“按值传送”，即把实在参量的值向形式参数作单向传送。形式参数的值的变化不会反过来影响实在参数的值。而对子程序则不然，在调用子程序时，将实在参数的存储单元的地址传送给子程序中的参数，因此，子程序的参数也利用了主程序实在变量的同一个存储单元，也就是，主程序的实参与子程序对应的虚参共用同一个存储单元。如果主程序实参为变量A，子程序形参为X，则A与X均在同一个单元，所以在子程序执行过程中，如果X的值发生变化，则主程序的变量A也随之变化。正是通过这个办法，把子程序中的数据传递回主程序中去。这种传递数据方法称为“按地址传送”。为便于理解，也可以将它理解为“换名操作”。即在调用时，将实在参数名代替形参名。例如，有一子程序：

```
sub F(a,b,c)
let c=a*a-b*b
end sub
```

如果主程序中有以下语句：

```
let m=5
let n=4
call F(m,n,t)
print t
```

将主程序中的m，n，t换子程序中的a，b，c。这样，子程序中的执行语句便变成

let t=m*m-n*n

运算后，得到t的值9，主程序中打印出：“9”。

m	n	t
5	4	9
a	b	c

但是，如果实在参数不是变量，而是常数或表达式，则无法采用上述的“按地址传送”或“换名操作”，而是象对函数那样，把常数和表达式的值传送到一个临时变量存起来，然后再把它的值传送到子程序（按值传送）。因此，子程序中形参值的变化，不会影响主程序中的表达式的值。例如CALL SUB (a,b+1)向子程序SUB传送a与b+1的值时，子程序能改变a的值不能改变b的值。由此可知，欲想将数据传回调用程序，调用程序的相应实参不应为表达式或常数，而只能用变量。

在例4.2.4中，在子程序COMB中，实参为变量M，N，C，D，F1，F2，F3以及表达式(M－N)，对M，N，D，F1，F2是“按地址传送”或“换名操作”，对表达式(M－N)则“按值传送”。子程序FACT和DIV中的形参FAC，QUOT的值通过F1，F2，F3，D，C传回调用程序。

子程序不仅可以接受简单变量形式的参数，也可以接受数组形式的参数。

当以数组作为形式参数时，应使用括号和逗号来标明数组的维数，如将一个一维数组表示为a( )，将一个三维数组表示为a(,,)。而相应的调用语句中的实参则只用数组名，不必加括号与逗号。

例 4.2.5 用数组作参数的例子。

```
! Set an array to all 1's.
!
dim A(3,5),B(2,10)
call Set_ones(A)      ! set A to all ones
call Set_ones(B)      ! and B
mat print A;B;
end
! An extenal subroutine.
!
sub Set_ones (A(,))          ! take any 2_dim arry
  for I=Lbound (A,1) to Ubound(A,1)
    for J=Lbound(A,2) to Ubound(A,2)
      let A(I,J)=1
    next J
  next I
end sub
```

```
Ok. run
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
Ok.
```

子程序中只指出数组的维数，各维的大小则由主程序中的 DIM 语句定义。如上述程序的子程序中使用一个二维数组参数。这个参数便可以传送任何大小的二维数组数据。这就是“可变数组”，即子程序中所用数组的大小不是固定不变的，而是随主程序而变的。这就使同一个子程序能处理不同大小的数组，大大提高了子程序的通用性。

子程序可以用通道号作参数(这一点与函数也不同，函数不允许用通道号作参数)，通道参数要被写成一个“#”后跟随一个 1～1000 之间的任意整常数。这个通道参数在子程序被调用时，用实参的通道号代替。

例 4.2.6 比较下面三个程序：

程序 1，用内部子程序。实参的通道号为#1：

```
! Demonstrate channel parameters.
!
```

```
sub A(#999,m)
   open #999: printer
   let x$ =""
   for i=1 to m
      let x$ =x$ &"x".
   next I
   print #999: "Sub……";x$
end sub
let n=10
call A(#1,n)
print #1: "Main……**********"
close #1
end
```

执行程序 1 后，在打印机上打印出：

```
Sub……XXXXXXXXXX            (子程序打印)
Main……**********            (主程序打印)
```

当主程序调用子程序 A 时，将通道号#1 传送给#999，即在子程序中实际上把#1 通道定义为打印机通道，并在子程序中打印出上述第一行。由于通道#1 在子程序中已被打开，所以在主程序中打印出上述第二行。

程序 2，用外部子程序：

```
! Demonstrate channel parameters
! in an external subroutine.
!
let n=10
call A(#1,n)
print #1: "Main……**********"
close #1
end
sub A(#999,m)
  open #999: printer
  let x$ =""
  for i=1 to m
     let x$ =x$ & "x"
  next i
  print #999: "Sub……"; x$
end sub
```

执行程序 2 后打印机输出

```
Sub……XXXXXXXXXX
Main……**********
```

打印的结果与程序 1 相同。

程序 3，用外部子程序，但没有使用通道参数：

```
! An external subroutine.
!
let n=10
call A(n)
★print #999: "Main……***********"
close #999
end
sub A(m)
  open #999: printer
  let x$=""
  for i=1 to m
    let X$=X$ &"X"
  next i
print #999: "Sub……": X$
end sub
```

```
Channel isn' t open.
```

执行程序 3 后打印机输出

Sub……XXXXXXXXXX

由程序 3 可以看出，通道号对各程序单位来说也是局部的。在不同程序单位中都使用 #999作通道号，它们并不代表同一个外部设备。它们应该分别打开和关闭。在程序 3 中，虽然主程序与外部子程序都使用了通道号 #999，但它们并不是指同一个通道。在子程序中，指定 #999 为打印机，但在主程序中并未用 OPEN 语句指定 #999 代表什么设备，因此，在主程序中，由于未打开 #999 而出现了错误。当主程序执行到"★"的语句时，给出"Channel isn't open."的信息，并把光标停在带"★"标志的程序行的标志位。

那么，程序 2 也是子程序 A，也是外部的，为什么不会出现上述情况呢？这是因为程序 2 中通道号是作为参数被传送的。由于使用参数沟通了两个程序单位的数据，因此在子程序中打开的 #1 通道(代表打印机)，在主程序中仍然有效。而在程序 3 中未将通道号作为参数传送。

4.2.4 递归子程序与递归程序设计

与递归函数一样，一个子程序直接或间接地由自己定义，则称其为递归子程序。

例 4.2.7 欧几里德算法的递归解。

```
! Program to find the greatest common divisor
! using euclidean algorithm with recursive.
!
input M,N
```

```
call GCD(G,M,N)
print G
end
sub GCD(G,A,B)
  if B=0 then
    let G=A
    exit sub
  else
    call GCD(G,B,Mod(A,B))
  end if
end sub
```

```
Ok. run
? 72,56↙
 8
Ok.
```

在这一过程中求最大公约数的过程由一个外部子程序 GCD(G,A,B)实现。其中 A,B 是该子程序的输入参数,G 是输出参数。即主程序用 M 给 A 传送数据,用 N 给 B 传送数据,子程序用 G 给主程序中的 G 传送数据。它的基本思想是,当除数为零时,被除数就是两个数的最大公约数;否则以除数作被除数,以余数作除数,再执行同样的过程。从描述上看这一过程好象是一个迭代过程,但从程序的形式上看,它并不是一个循环结构。所以说,递归使得那些似乎是重复的过程,不用循环处理。反过来说,也有可能用循环迭代来替代递归。如何用迭代替代递归的问题,以后再讨论。

递归是一种有效的思考问题的方法和有力的数学工具。把递归的概念引入程序设计,也给程序设计带来生机。在 4.1.4 节中已经指出,许多问题用递归过程来描述不仅非常自然(如人工智能中的许多问题),而且其正确性证明也比较容易。这就为程序设计提供了许多方便。递归几乎已经成为一种与逐步细化和结构化程序设计相平行的一种独立的程序设计方法了。

那么,如何进行递归程序设计呢?通过 4.1.4 节和本节前面所举的简单例子可以看出递归程序有这样两个特点:

(1) 在递归调用过程中,每调用一次都要生成一组新的"量"。这些新的量与原来的"量"具有相同的名字,但却代表了不同的含义。如例 4.2.7 中的 GCD(G,A,B)中的 B,与 GCD(G,B,MOD(A,B))中的 B 虽然具有相同的名字,但它们在括号内参数的顺序不同,因此,作用不同,前者代表的是除数,后者代表的是被除数。并且每递归调用一次,就要由上一层向下一层传递一次数据。例如,例 4.2.7 中,如果 B≠0,则要将子程序中的 B 传给下一层 GCD 中的 A,将 A 除以 B 的余数传给下一次调用的子程序的参数 B,每次递归调用都要进行参数替换、环境保护等工作。为了避免混乱,应当使用局部量。显然,采用外部函数与外部子程序是很合适的:要在递归调用中传递的量数,就把它定义成参数,其余的量则作为局部变量。例 4.2.7 中主程序的 M,N 作为参数传递到子程序,G 是由子程序传回主程序的结果,关系十分清楚。A,B,G 是局部变量。因此,进行递归程序设计的关键是要确定哪些量作为参数

传递。

(2) 递归将引起回头进行一系列的链式操作。这种链式操作在一定的条件下将结束。或者说，任何有意义的递归过程都是有限的，或称有边界的。如例 4.2.7 中的

```
if B=0 then
    let G=A
```

便是它的边界条件。一个递归程序若无边界条件，最后将会经过多次调用后，产生内存无法存储的局部变量的错误，显示出"Out of memory"的信息，非正常地结束运行。这是因为每次递归调用时，必须记录每次产生的数据，以便"返回"时使用(这叫保留现场，或环境保护)，需要占一定内存单元，递归次数太多，占内存会很大。所以，进行递归程序设计的第二个要点是，要正确地设定递归的边界条件。

递归的概念源于数学。所以许多人认为递归只适用于计算。其实，递归的应用范围要广泛的多。下面就以几个在非数值领域内应用的例子，来说明递归程序设计的方法。

例 4.2.8 在数组 A(1 : N)中检索 X，若在 A(1 : N)中有一个元素 A(K)=x，则返回 K，否则返回"0"。

(1) 主程序：本题可以归纳成如下四个功能：

(i) 说明及定义；

(ii) 输入数据及给定 x；

(iii) 查找 x；

(iv) 输出。

程序如下：

```
! Search X in an array by secursive.
!
dim A(10)
data 6,7,5,3,8,2,9,1,4
input prompt "How many items (<=9)? ": N
mat read A(N)

input prompt "The X is:": X
call SEARCH(A,1,N,K,X)
print
if K=0 then
  print "The item can't be found."
else
  print "The item is A(";K;")."
end if
end
```

(2) SEARCH 子程序设计：SEARCH 子程序的功能是在数组 A(1 : N)中找 X。当数组 A 中的元素无序时，只能进行穷举查找。查找结束的条件是：

(i) X 不在 A 中。

(ii) 在 A 中找到了 X。

当采用递归方法查找 X 时，上述两条就是递归的边界条件。

递归查找过程如图 4.5 所示。方法是从 A 的第一个元素起顺序查找。设某次查找从第 H 个元素开始找起，即在数组 A(H:N)中找 X 的位置 K，把它写为 SEARCH(A，H，N，K，X)。这里 A，H，N，X 是子程序 SEARCH 的入口参数，K 是它的出口参数。

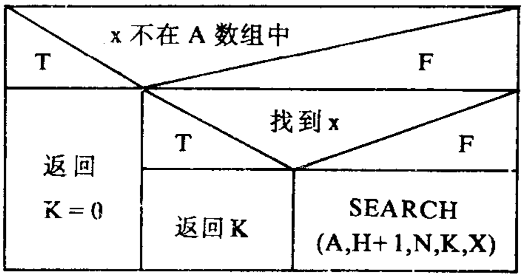


图 4.5

那么如何描述条件“X 不在 A 中”和“找到了 X”呢？

(i) 一开始令 H=1，每执行一次 SEARCH，要找 A(H)是否等于 X，如果不是，则令 H=H+1 再找 A(H+1)是否等于 X。当 H=N 时，查 A 中的最后一个元素 A(N)是否等于 X；当 H>N 时，便是“X 不在 A 中”。

(ii) 每次查找都要检查 A(H)是否等于 X，若 A(H)=X，则应将 X 在数组中的位置 H 返回调用程序，执行 K=H 的操作，通过 K 将此值返回。

根据上述分析可以写出下面的 SEARCH 子程序：

```
sub SEARCH(A( ),H,N,K,X)
  if H>N then
    let K=0
    exit sub
  else if A(H)=X then
    let K=H
    exit sub
  else
    call SEARCH(A,H+1,N,K,X)
  end if
end sub
```

程序运行结果如下：

```
Ok. run
How many items (<=9)? 9↙(共有数据 9 个)
The X is: 6↙(要找的数为 6)

The item is A(1). (结果是第 1 个元素)
Ok. run
How many items (<=9)? 9↙
The X is: 4↙

The item is A(9).
Ok. run
How many items(<=9)? 8↙
The X is: 3↙
```

```
The item is A(4).
Ok.
```

(3) 讨论：上面使用外部子程序 SEARCH 来实现查找。数组 A 在主程序与子程序之间以及子程序之间反复传送，但并没有对数组 A 作任何变动。但是又不能不把 A 作为参数。因为外部子程序与主程序是两个独立的程序单位，不把 A 作为参数传送，将在子程序中出现数组没有定义的错误。但是，数组传递是要花时间的，尤其是数组规模较大时，效率就低了，这种情况下，也可以使用内部子程序。

程序 2：

```
! Search a X in an array by secursive.
!
dim A(10)
data 6,7,5,3,8,2,9,1,4

sub SEARCH(H,N,K,X)
  if H>N then
    let K=0
    exit sub
  else if A(H)=X then
    let K=H
    exit sub
  else
    call SEARCH(H+1,N,K,X)
  end if
end sub

input prompt "How many items (<=9)? ":N

mat read A(N)
input prompt "The X is:":X

call SEARCH(1,N,K,X)
print
if K=0 then
  print "The item can't be found."
else
  print "The item is A(";K;")."
end if
end
```

```
Ok. run
```

```
How many items (<=9)? 9↙
The X is:6↙

The item is A(1).
Ok. run
How many items (<=9)? 9↙
The X is: 4↙

The item is A(9).
Ok. run
How many items (<=9)? 9↙
The X is: 3↙

The item is A(4).
Ok.
```

由于查找结果只给出一个值，程序 2 中的子程序的功能也可以用一个内部函数来实现。程序可以改写如下：

程序 3：

```
! Search X in an array by secursive.
!
dim A(10)
data 6,7,5,3,8,2,9,1,4

def SEARCH(H,N,X)
  if H>N then
    let SEARCH=0
    exit def
  else if A(H)=X then
    let SEARCH=H
    exit def
  else
    let SEARCH=SEARCH(H+1,N,X)
  end if
end def

input prompt "How many items (<=9)? ": N

mat read A(N)
input prompt "The X is:": X

let K=SEARCH(1,N,X)
```

```
print
if K=0 then
  print "The item can't be found."
else
  print "The item is A(";K;")."
end if
end
```

```
Ok. run
How many items (<=9)? 9↙
The X is: 5↙

The item is A(3).
Ok. run
How many items (<=9)? 9↙
The X is: 3↙

The item is A(4).
Ok. run
How many items (<=9)? 9↙
The X is: 8↙

The item is A(5).
Ok.
```

程序 2 与程序 3 中，把递归调用过程中需传送的量作为参数，使其成为局部量，把不需传送的量(如数组)作为全局变量，供主程序与子程序共同使用。在这种情况下，全局变量并没有什么不好，只不过必须小心行事，不要造成各模块之间的数据干扰就可以了。

前面给出了使用外部子程序的例子，下面再来讨论对于这同一问题能否使用外部函数来解决。

我们已经知道：使用外部子程序与外部函数时，在各程序单位中要使用的信息，都必须全部使用参数传送。使用外部函数时，应在主程序中引用外部函数之前用 DECLARE DEF 语句对其加以说明。请看下面的程序：

程序 4：

```
! Search X in an array by secursive.
!
dim A(10)
data 6,7,5,3,8,2,9,1,4
declare def SEARCH

input prompt "How many items (<=9)?":N
```

```
mat read A(N)
input prompt "The X is:": X

let K=SEARCH(1,N,X)

print
if K=0 then
  print "The item can't be found."
else
  print "The item is A(";K;")."
end if
end

def SEARCH(H,N,X)
  if H>N then
    let SEARCH=0
    exit def
 ★else if A(H)=X then
    let SEARCH=H
    exit def
  else
    let SEARCH=SEARCH(H+1,N,X)
  end if
end def
```

运行上述的程序后，系统给出如下提示：

This must first appear in a DIM or DECLARE DEF.

然后把光标停在有"★"标记的程序行。要求用 DIM 语句或 DECLARE DEF 语句明确A(H)是数组元素还是函数。因为在主程序中，虽然已定义了数组 A(1：N)，但它并没有传到子程序去。在子程序中没有定义过 A 数组，因此无法辩别 A(H)是数组元素还是函数调用(这两种表示形式是一样的)。事实上，子程序中的 A(H)是一个数组元素并且是局部的，因为外部函数是一个独立的程序单位。但可不可以在函数中加一个 DIM 语句呢？请看下面的程序：

程序 5：

```
! Search a X in an array by secursive.
!
dim A(10)
data 6,7,5,3,8,2,9,1,4
declare def SEARCH

input prompt "How many items (<=9)?": N

mat read A(N)
```

```
input prompt "The X is:": X

let K=SEARCH (1,N,X)

print
if K=0 then
  print "The item can't be found."
else
  print "The item is A(";K;")."
end if
end

def SEARCH(H,N,X)
★ dim A(9)
  if H>N then
    let SEARCH=0
    exit def
  else if A(H)=X then
    let SEARCH=H
    exit def
else
let SEARCH=SEARCH(H+1,N,X)
end if
end def
```

```
Ok. run
How many items (<=9)? 9↙
The X is: 6↙

The item can't be found.
Ok. run
How many items (<=9)? 9↙
The X is: 4↙

The item can't be found.
Ok. run
How many items (<=9)? 9↙
The X is: 3↙

The item can't be found.
Ok.
```

在子程序中有"★"的标志处给出了一个 DIM 语句，把 A 定义为一维数组。但程序执行

后，却找不到任何一个本来应能查到的数。这是什么原因呢？问题在于，虽然在函数中定义了数组，而且这个数组与主程序中的数组同名，但是这外部函数中的 A 数组是局部变量，与主程序中的 A 数组没有关系。它们各自独立，都是本程序单位中的局部量。在外部函数中的这个数组并没有被输入任何元素。要使这个程序能正确地执行，还须将数组作为参数传送。当然，这时子程序中也不允许再给数组以定义了。因为在 DEF 语句中已用"(　)"已经说明 A 是一个数组了，再使用 DIM 语句将会造成"重复定义"的错误。请看下面的程序：

程序 6：

```
! Search a X in an array by secursive.
!
dim A(10)
data 6,7,5,3,8,2,9,1,4
declare def SEARCH

input prompt "How many items (<=9)?": N

mat read A(N)
input prompt "The X is:": X

★ let K=SEARCH(A,1,N,X)
print
if K=0 then
  print "The item can't be found."
else
  print "The item is A(";K;")."
end if
end

★ def SEARCH(A( ),H,N,X)
  if H>N then
    let SEARCH=0
    exit def
  else if A(H)=X then
    let SEARCH=H
    exit def
  else
★ let SEARCH=SEARCH(A,H+1,N,X)
  end if
end def
```

```
Ok. run
How many items (<=9)? 9↙
The X is: 6↙
```

```
The item is A(1).
Ok. run
How many items (<=9)? 9↙
The X is: 4↙
The item is A(9).
Ok. run
How many items (<=9)? 9↙
The X is: 3↙
The item is A(4).
Ok.
```

程序 6 又给出了与本题第一个程序同样的正确答案！但是，要指出的是这两个程序的效率是不相同的。因为传送数组给外部函数时，外部函数每被调用一次，整个数组要被传送到一个临时数组中。所以程序 6 运行时，要比程序 1 占用更多的存储空间，且执行速度要慢。请注意：对外部函数来说，参数传递的方式是“按值传送”，用临时变量存储实参的值，因此多占空间。而对子程序来说，是实参和形参共同存储单元，比用外部函数占空间少。

例 4.2.9 快速排序(QKSORT)：在 3.4.3 节中，已介绍了一种常用的交换排序方法——起泡排序。快速排序也称“分区交换排序”，是由著名计算机科学家 C. A. R. Hoare 在起泡排序的基础上提出的一种高效率的排序算法。它的基本思想是：在待排序数列中选取一个元素(通常是第一个元素)X，按某一规律经过多次交换位置后，它移到某一位置，以此位置对原数列进行划分，使得得到的两个子数列对 X 来说符合排序规律(如按递增排序时，划分后 X 前面的子数列中的每个元素都不大于 X，X 后面的子数列中的每个元素都不小于 X)。元素 X 称为该数列的划分元素。接着按此方法对两个子数列再划分，直到得到不需再进一步划分的子数列为止。显然，这一排序过程具有明显的递归性。

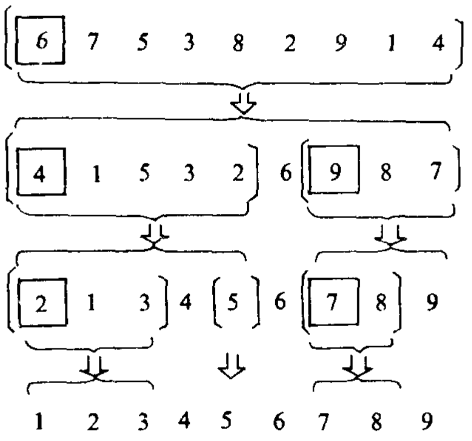


图 4.6

图 4.6 是一个用快速排序法进行排序的实例。图中方括号中是待排序的数列。方框中的数字是每次选定的划分元素。

下面讨论如何设计快速排序程序。

首先，可以把该问题概括为如下四部分内容：

(i) 说明及定义；

(ii) 输入待排序数列；

(iii)用 QKSORT 方法进行排序；

(iv) 输出排好序的数列。

S1：说明及定义
S2：输入数据
S3：QKSORT
S4：输出

图 4.7

对问题从整体上的概括，便形成主模块的算法如图 4.7 所示。

凭已有的经验对主模块中的 S1，S2，S4 直接编码是毫无问题的。这里主要考虑如何获

得子算法 QKSORT。

设某一时刻要对数列 A(S：T)进行排序，要给子程序 QKSORT 传递的是 A，以及它的下标的上、下界 T 与 S，要求子程序 QKSORT 输出一个已排好序的数组 A。因此主程序中调用子程序的语句可以写成：

call QKSORT(A,S,T)

的形式调用。

主程序如下：

```
! Sorting with QKSORT.
!
dim A(9)
data 6,7,5,3,8,2,9,1,4

input prompt "How many items?": N
mat read A(N)

let S=1
let T=N
call QKSORT (A,S,T)

print "Sorted sequence is:";
mat print A;
end
```

下面再来考虑如何设计子程序 QKSORT。

由前面的介绍可知，使用快速分类法进行排序时，要先进行划分。设某一次对数列 R(S：T)即 R(S)，R(S+1)，…，R(T−1)，R(T)进行划分，划分后，划分元素的位置为 I，则可以产生如下三个序列：

Seq1：R(s)，R(S+1)…，R(I−1)；

Seq2：R(I)；

Seq3：R(I+1)，R(I+2)，…，R(T−1)，R(T)。

其中 R(I)的位置已经最后确定。因此每经过一次划分后，要分别对 Seq1 与 Seq3 再递归地执行 QKSORT。当然，每次划分是有条件的，即所划分的数列必须有两个以上元素。一个以下的元素无需再进行排序。

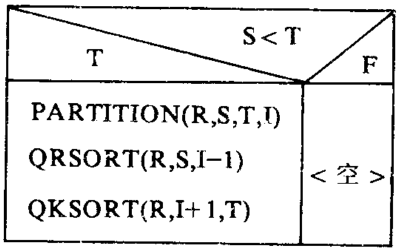


图 4.8

根据上述分析可以得到图 4.8 所示的 QKSORT 子算法。R 是一维数组，S，T 是下标下界和上界，I 是划分元素的位置。PARTITION 子程序的作用是找 I 的值。对该算法编码，可得到如下 True BASIC 子程序。它是一个递归子程序。该子程序中还没有精确描述的是

```
sub QKSORT (R(),S,T)
```

```
  if S<T then
    call PARTITION(R,S,T,I)
    call QKSORT(R,S,I-1)
    call QKSORT (R,I+1,T)
  end if
end sub
```

PARTITION,即划分子程序。下面以图 4.9 为例来说明一趟划分的基本方法:

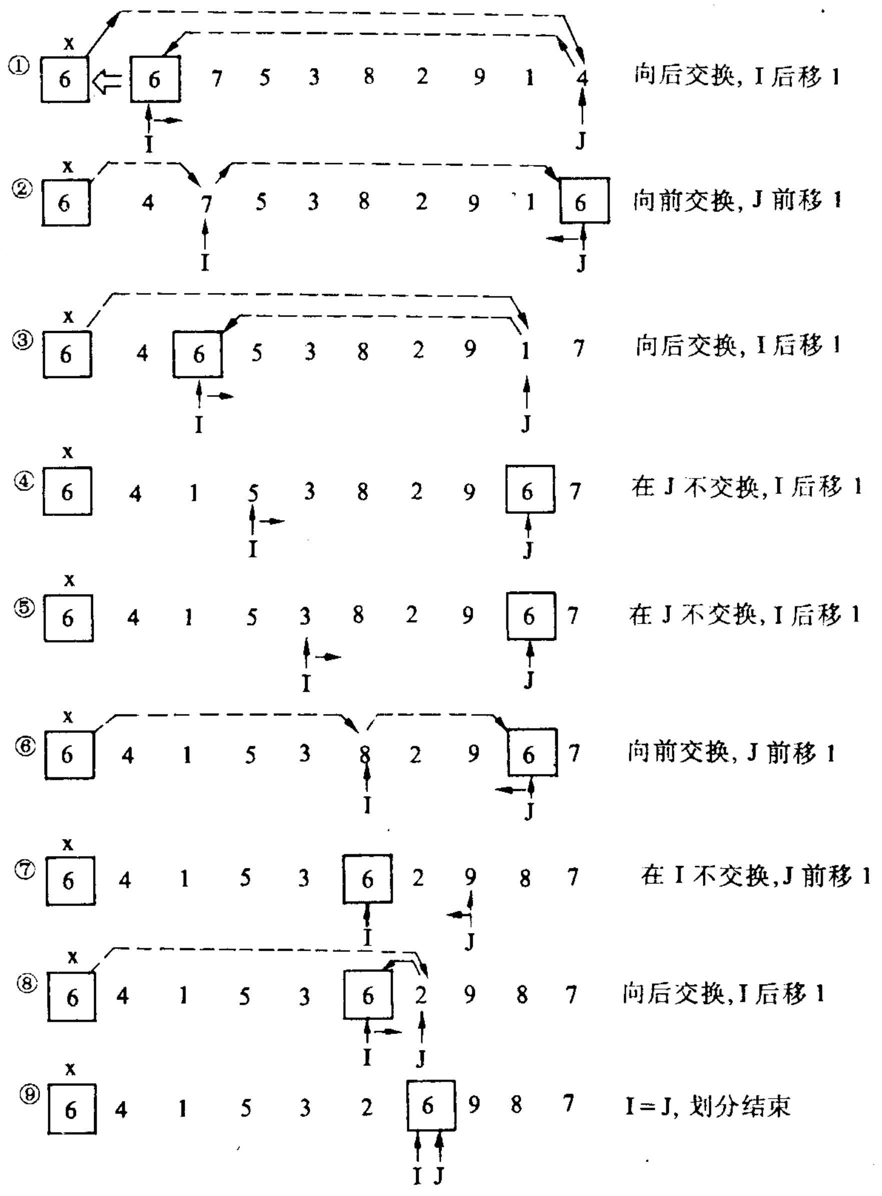


图 4.9 一趟划分示例

为了对数列 R(S),…,R(T)进行划分,要设置两个指针,让 I 指向 R(S)(或者说令 I=S),让 J 指向 R(T)(或者说令 J=T),并且取 R(S)⇒X(即 I 所指元素)作为划分元素。接着反复执行如下操作:

① 将 R(J)与 X 相比,若 R(J)>X,就将 J 往左移(即 J=J-1),直到移到 R(J)<X 处时,交换 R(I)和 R(J),由于此时 R(I)=X,因此也可以表示为:将 R(J)⇒R(I),将 X⇒R(J)

并令 I=I+1。例如，第一次将 R(9)⇒R(1)，将 X⇒R(9)，即将 4 移到第一个位置，将 6 移到第 9 个位置。见图 4.9②。经交换后，J 右面的元素都比 X 大，而 I 以左的元素都比 X 小。这时尚未划分好的元素在 I 与 J 之间，且划分元素处于 J 处，即新的 R(J)=X。

② 接着将 R(I)与 X 比较，若 R(I)<X，就将 I 向右移(即 I=I+1)，直到移到 R(I)>X 处时，交换 R(I)和 R(J)，由于此时 R(J)=X，因此，也可以表示为：使 R(I)⇒R(J)，X⇒R(I)。图 4.9②中，当 I=3 时，R(I)>X，即 7>6。将 R(3)⇒R(9)，即将 7⇒R(9)，X 的值 6⇒R(3)，见图 4.9③，并令 J=J−1。未划分的区间进一步缩小。

反复执行上述操作直到 I=J，便完成了一趟划分。可以看出，划分元素(本例为 6)在序列的左面和右面两端反复地“跳跃”，每次都向中部靠拢，直到到达它的最后位置为止。注意，X 的值始终不变(等于 6)。

在上述划分过程中，每一步都应判断不能使 I>=J。如对序列

[9] 6 7 8 3 2 1 5 4

进行划分时，选 9 为划分元素，若不规定只有 I<J 时才移动 J 指针，那么 I 最后会左移 9 的左边。所以当划分元素在左端时，J 左移的条件是：“R(J)>X and J>I”；而当划分元素在右端时，I 右移的条件是“R(I)<X and J>I”。

根据上述分析，可以得到划分子算法如图 4.10 所示。根据图 4.10 所示划分子算法，可以写出划分子程序如下：

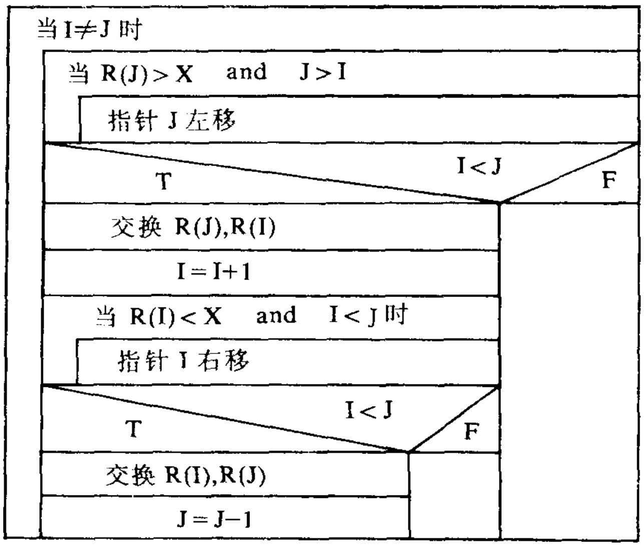


图 4.10

```
sub PARTITION (R(),S,T,I)
  let I=S
  let J=T
  let X=R(S)
  do while I < >J
    do while R(J)>X and J>I
      let J=J−1
    loop
```

```
        if I<J then
          let R(I)=R(J)
          let R(J)=X
        else
          exit sub
        end if
        let I=I+1
        do while R(I)<X and J>I
          let I=I+1
        loop
        if I<J then
          let R(J)=R(I)
          let R(I)=X
        else
          exit sub
        end if
        let J=J-1
      loop
    end sub
```

程序的运行结果如下：

```
Ok. run
How many items? 9↙
Sorted sequence is: 1 2 3 4 5 6 7 8 9
Ok. run
How many items? 8↙（只取 data 语句中前 8 个数据）
Sorted sequence is: 1 2 3 5 6 7 8 9 0
Ok. run
How many items? 7↙
Sorted sequence is: 2 3 5 6 7 8 9 0 0
Ok. run
How many items? 6↙
Sorted sequence is: 2 3 5 6 7 8 0 0 0
Ok. run
How many items? 5↙
Sorted sequence is: 3 5 6 7 8 0 0 0 0
Ok. run
How many items? 4↙
Sorted sequence is: 3 5 6 7 0 0 0 0 0
Ok.
```

由于程序中已定义 A 数组有 9 个元素，而且主程序中最后的打印语句要求打印出数组的全部元素，因此，对未读入数据的数组元素，打印出其初值零。

例 4.2.10 Hanoi 塔问题：传说印度教的主神梵天创造世界时，在印度北部佛教圣地贝拿勒斯圣庙里，安放了一块黄铜板，板插着三根宝石针，在其中一根宝石针上，自下而上地放着由大到小的 64 个金盘。这就是所谓的梵塔(Hanoi)，如图 4.11 所示。梵天要僧侣们坚持不渝地按下面的规则把 64 个盘子移到另一根针上：

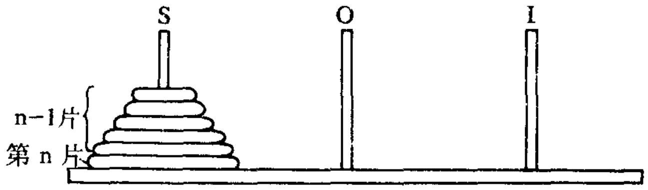


图 4.11 Hanoi 塔

(1) 一次只能移一个盘子；

(2) 盘子只许在三根针上存放；

(3) 永远不许大盘压小盘。

梵天宣称，当把他创造世界之时所安放的 64 个金盘全部移到另一根针上时，世界将在一声霹雳声中毁灭。那时，他的虔诚的信徒都可以升天。

要求设计一个移盘子的 True BASIC 程序。为了使问题具有普遍性，设共有 N 个金盘，并把问题抽象：把原先放在 S(Source)针上的 N 个盘子，经中间针 I(Indirect)，最后移到目的针 O(Objective)上，并用算法 Hanoi(N,S,I,O)表示。

把这一描述具体化有两种考虑方法：一种考虑着眼于上层的小盘一个一个地如何移；另一种考虑着眼于移动底层的大盘。采用后一种考虑方法，即先设法把上面的(N－1)个盘子从 S 针移到 I 针，然后将第 N 个盘子从 S 针移到 O 针。这样就把移动 N 个盘子从 S 到 O 针的问题转化成：①从 S 把(N－1)个盘子移到 I；②再从 S 将第 N 个盘子移到 O 针；③再将上面(N－1)个盘子从 I 移到 O。第②步是很简单的。第①、③两步实质上是一样的，只是针不同而已，为了实现上述算法，必须解决如何把(N－1)个盘子从 S 移到 I 和从 I 移到 O。以由 I 移到 O 为例说明。按上述同样的办法，可以把(N－2)个盘子从 I 先移到 S，再把第(N－1)个盘子移到 O。如此递归地进行，每一次都把问题转化为同样性质但“规模”小一级的新问题，直到第 0 级(无盘可移)为止，问题便可以圆满解决。

上述算法 Hanoi(N,S,I,O)可以形式地描述为

(1) 当 N＝1 时，可以直接把盘从 S 针移到 O 柱，记作 Move (S,1,O)；

(2) 当 N＞1 时：

① 将 S 上的(N－1)个盘借助 O 移到 I，记作 Hanoi(N－1,S,O,I)；

② 将 S 上的第 N 个盘移到 O，记作 Move (S,N,O)；

③ 将 I 上的(N－1)个盘借助 S 移到 O，记作 Hanoi(N－1,I,S,O)。

子程序 Move (S,N,O)表示“把第 N 个盘子由 S 针搬到 O 针”。

由上述分析，并设 S 针为 1＃，I 针为 2＃，O 针为 3＃，很容易得到下面的 True BASIC 程序：

```
! Program of Hanoi tower problem.
!
```

```
input prompt "How many disks?": N
let S=1
let I=2
let O=3
call Hanoi (N,S,I,O)
end

sub Hanoi (N,S,I,O)
  if N=1 then
     call Move (S,1,O)
  else if N>1 then
     call Hanoi (N-1,S,O,I)
     call Move (S,N,O)
     call Hanoi(N-1,I,S,O)
  end if
end sub

sub Move(S,N,O)
  print "Move disk"; N;
  print "from"; S; "to";O
end sub
```

```
Ok. run
How many disks? 3
Move disk 1 from 1 to 3
Move disk 2 from 1 to 2
Move disk 1 from 3 to 2
Move disk 3 from 1 to 3
Move disk 1 from 2 to 1
Move disk 2 from 2 to 3
Move disk 1 from 1 to 3
Ok.
```

4.3 库 文 件

4.3.1 库文件的概念

外部函数和外部子程序中使用的全部变量都是局部的。它只能通过参数与调用程序接口。每个外部函数或外部子程序都是一个独立的程序单元，它可以被独立地编写，独立地调试；只要接口关系正确，可以被任何需要它的程序调用。

为了能更方便地使外部函数和外部子程序能为不同的程序共享，True BASIC 允许用库文件的形式保存一些经常使用或专用的外部函数和外部子程序。

一个库(Library)是存储在一个文件(file)中的外部函数和外部子程序的集合。一个子程序或函数被收到库中后,任何一个程序都可以从库中调用所需的函数或子程序。

库文件不包含主程序,所以库文件是不能被执行的。库文件可以是已被编译好的函数或子程序,也可以存放未经编译的用 True BASIC 语言编写的程序单位。当把一些经常使用或专用的库文件编译后,使用它时便会感到更方便。

如果把主程序比做机车头,那么库文件中的各个函数和子程序就好比是车库中的车厢。只要需要,便可以将它通过接口挂在任何一个合适的车头后面。

True BASIC 除允许用户以库文件的形式存放自己定义的函数和子程序外,还提供了四个包含数学函数的库,一个包含图画的库,一个包含使用菜单子程序的库。

4.3.2 库文件的形成与使用

1. 库文件的形成:在几个无主程序的外部函数和外部子程序前面加上一条 EXTERNAL 语句,然后用 SAVE 命令将它们一起存入磁盘,便形成一个库文件。SAVE 命令的格式为

SAVE〈文件名〉

True BASIC 文件名的命名规则已在 2.1.4 节的 5 中介绍。

下面是 DOS 2.0 操作系统的保留名,不能把它们作为一般文件名来使用:CON.(键盘/屏幕控制),AUX,COM1.(第一个异步通信适配器接口),COM2.(第二个异步通信适配器接口),LPT1.,PRN.(第一台并行打印机接口),LPT2.(第二台并行打印机),LPT3.(第三台并行打印机),NUL.(不存在设备)。

在一个库文件中可以包含多个外部子程序和外部函数,但不允许有同名的子程序或函数。

2. 库文件的使用:当程序要调用库中的子程序或函数时,应在调用程序中,在调用之前使用一条LIBRARY 语句,在 LIBRARY 语句中指出要检索的库文件的名字。格式如下 。

LIBRARY"〈文件名〉","〈文件名〉",……

这样,程序便可以用 CALL 语句调用放在库文件中的子程序,或用表达式调用存在库文件中的外部函数了。用起来好象这些外部子程序与外部函数是跟在该程序的 END 后面一样。这样可以免去程序设计者自己去设计或抄写每一个外部子程序或外部函数。尤其在实现比较复杂的算法时,利用库文件的好处更为明显。

4.4 模块化程序设计

4.4.1 设计大程序的策略——模块化

把两根线接起来是很容易的,但是,要把许多根揉在一起的线接成一根时,将会发现,主要精力不是花在简单地去接一根一根的线,而是花在去理出每条线的头绪来。程序设计也是如此。

人们学习程序设计，都是从小规模的程序开始的。只有学会设计小程序，才有可能开始学习设计大程序。但是只凭设计小程序的经验去设计大程序，却是很不够的。一个大程序常常包含了多个任务，每个任务中都要涉及一些变量或数组。当程序的规模大到一定的程度时，多如繁星的变量和数组，穿梭往返的流程，使人眼花缭乱，即使有经验的程序员也常常会顾此失彼，甚至错误百出。

还经常发现，给学生讲太容易的东西时，引不起学生的兴趣。学生的听课兴趣会随着所讲内容的增加而提高。然而，当所讲的内容多到一定程度时，学生们听课的兴趣反会陡然下降，甚至一点儿也没听进去。这是因为每个人的智力，特别是对详细的信息处理能力是有限的。问题的规模太大、内容过多、解题的路径太长等将会引起人的脑力的疲劳，甚至不能接受。

人们解决大问题的一种有效策略是"分而治之"。当按照一定的规则，把一个大问题分割为人的智力能及的一些小问题时，整个问题将变得容易解决了。或者说，问题的复杂程度降低了。这一规律，可作如下形式化的描述：

设 C(P)是问题 P 的复杂性函数，E(P)是解决问题 P 所需要的工作量。若能把问题 P 分解为 P_1，P_2 两个子问题，即 $P=P_1+P_2$，则有

$$C(P)>C(P_1)+C(P_2)$$

以及

$$E(P)>E(P_1)+E(P_2)$$

人们处理复杂问题时常常采用这种策略：…如为了制定全市的规划，先把它划分为若干方面的任务(工业、商业、市政、交通、文教卫生、……)，先分别制定出部门规划，然后再将它们综合为全市的规划。再如，全国人口普查，也是采用先分地区，块块统计，然后汇总的方法。否则，将无法完成这些任务。

模块化程序设计是分而治之策略在程序设计中的应用。它要求遇到一个大程序时，应把它分割成为人们的智力能及的一些程序模块。

前面介绍了函数与子程序。这是两种分解程序的重要手段。当把程序分解成一些程序模块后，提高了程序的编写效率和程序的清晰度，使人们对每一个模块都能考虑得很周全，容易保证程序的正确性。从结构化程序设计的角度看，函数和子程序反映了程序的两级抽象：在调用一级，人们只需关心它能"做什么"，至于"怎么做 "的细节可以暂时不考虑，在定义一级再去考虑"怎么做"的细节。

定义模块是模块化程序设计的关键。只有每个模块的概念明确时，整个程序才具有较好的可理解性。以数据为中心和以功能为中心是得到概念明确的模块的两种基本方法。前者是把针对某一特定的数据集(如文件，数据库模式以及数组等)的操作划为一个模块，这时各模块可以定义为××数据集的操作。后者是按功能划分，即把整个程序划分为一些相对独立的功能，每个模块最好只完成一项即明确又单纯的功能。如在例 4.2.4 中，把计算 C_m^n 的过程，划分为 IN，COMB，OUT 等三个模块，这三个模块的概念是很明确的。当其中的某些模块还比较复杂时，还应再进一步分解。

模块化设计的基本原则是努力提高模块的独立性。模块的独立性可以用模块的强度(也称内聚——Cohesion)和耦合(Coupling)性两个尺度进行衡量。

模块的强度是指一个模块内各语句间互相关联的紧密程度。按以功能为中心的模块划

分方法，希望一个模块内包括且仅包括完成某一具体任务的所有成份。或者说，当一个模块恰好做一件事时，如求 C_m^n 时划分为 IN，COMB，OUT，模块的强度就比较强。这种模块强度称为功能强度(fuctional slregngth)。在程序设计时常见的一种毛病是，在一个模块中包含了多个功能，而有些语句是为多个功能所使用。这就增加了模块的模糊性。这种模块称为巧合强度模块，它可读性低，有牵一发而动全身之患，使程序难以修改和维护。

耦合是指模块间相互联系的紧密程度。在例 4.1.7 中，举了一个埋藏有错误的例子，两个模块(子程序与主程序)中使用了全局变量，降低了程序的可读性，也是埋藏错误的根源。这种耦合称为公共耦合，是一种较差的耦合。在 4.2.3 中，已经说明，局部变量的使用，使两个模块间仅通过参数传递数据。这种耦合称为数据耦合，是一种较好的耦合关系。另外，在例 4.2.1 中，子程序 Subr 与主程序间没有数据交换，它只受主程序的调用。这种耦合称为非直接耦合，是一种最好的耦合关系，即模块间联系最弱。

模块的内聚性高、耦合性弱，模块的独立性就高。

提高模块内聚性，降低耦合性的有力武器是抽象和信息屏蔽。抽象能使各模块具有清晰的概念，符合程序设计中的“逐步细化”的开发策略，例如子程序就反映了程序的两级抽象。就是前面提到的调用程序解决“做什么”，子程序才解决“如何做”，并具体给出实现的细节。信息屏蔽使得包含在模块内的信息(过程和数据)对于不需要这些信息的其它模块是不可存取的。这样如果一个模块有错误，也难以传播到其它模块。每个模块都能独立地理解、编写、测试、维护，而不受其它模块的影响。

总之，模块化程序设计方法的应用，把整个问题都局部化了，问题的复杂程度也降低了。一个大程序分为若干个模块交给不同的人分别去完成时，每个人所承担的一部分任务都是自己的智力容易解决的。这就提供了一种按照“工程化”方式来“生产”程序的有效途径，并且能从以下几个方面保证程序的质量。

1. 易读性：为使模块具有高的独立性，各模块都应有一个清晰而单一的功能。一个原来复杂而含混的问题被划分为一个个清晰的部分，就象一篇章、节、段落清晰的文章一样，能使读者很快理清文章的思路与内容。

2. 可修改性：信息屏蔽原则的使用，使得当一个程序所实现的任务需作部分调整或因发现某一错误而对程序进行调整或修改时，只需修改一个或少数模块，不致造成涉及全局的修改。当程序所承担的功能作较大调整时，还可以更换某些模块，不致造成整个程序的废弃。这种可修改性，是提高程序可维护性的重要保证。

3. 可验证性：人们可以单独地验证一个模块的正确性，即脱离程序的上、下文也能用人工方法(静态检查)或上机的方法(动态检查)验证该模块的正确性(当然上机调试时应当根据接口的规定，给出必要的驱动模块和承接模块)。

模块的独立性还提供了共享性，在扩充已有系统，或建立新的系统时，人们可以充分利用现有的模块，用积木式的方法进行开发。如 True BASIC 提供的库文件，能把那些经过验证是正确的子程序和函数，经编译保存在库文件中，需要时可以直接调用。这样做不仅节省了人力，而且由于采用了被验证正确的模块避免了新编程序易出错的那种可能性。

一般地说，模块划分得足够小时，每个模块所包含的复杂性也会足够地小，开发模块所花费的代价也就会降低。但是随着模块划小，模块数量增加，模块间的联系(模块间传递信息)加强，处理接口的工作量也就增加了。到了一定程度，总的开发代价而会随着模块不断划

小而大大增加，见图 4.12 所示。

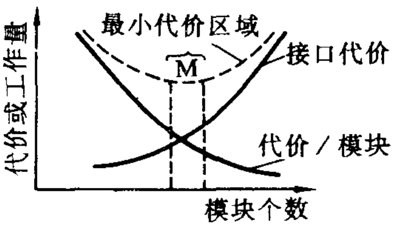


图 4.12

因此，在模块化程序设计中，存在一个最小代价问题。通常认为，每一个模块以 50 行左右为宜，打印出来不超过一页，以便于阅读。最多也不应超过 200 行；还有人主张把模块大小限制在“三个人日之内能完成（编码）”，但这些都不是绝对的。因为模块的划分主要是依据独立性原则。

4.4.2 模块间的层次结构

采用模块化方法所得到的系统是由互相连结的模块组成的。按什么样的结构组织各模块是模块化程序设计的另一个关键问题。它与模块的划分方式密切相关。早期的模块化程序设计方法不限制模块的划分方式，使系统呈现出图 4.13 所示的“嵌镶”式结构。这种结构中，各模块之间存在着很密切的联系，随着把大的问题逐次划小，模块间的关系越来越复杂。

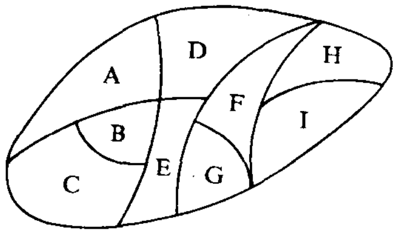


图 4.13

结构化设计方法出现以后，模块化程序设计被赋予结构化的新含义，要求在设计程序系统时按层次组织模块。在例 4.2.4 中，已经给出求 C_m^n 的层次结构。层次结构可以用树（例挂树）型结构来描述，如图 4.4 所示。

按层次关系描述的系统结构中，处在低一层的模块是对其上一级所提出的任务的进一步表达。即最上层的模块只是抽象地提出问题，它只说明“做什么”。当向下细化时，由抽象逐渐变为具体。最底层，才是对“怎么做”细节的描述。从上层到底层细化的过程，是愈来愈精细地描述“怎么做”的过程。这种抽象的方法，可以使人们在某一抽象级上集中地考虑该层模块的结构，作宏观的安排，不必急于考虑具体细节。这样，可以保证程序逻辑的正确、结构的完善，并且把出错的可能控制在较小的范围内。

所以结构图是自顶向下逐层展开的，它既能反映系统的总体，又能反映系统的细节。

结构图应按如下原则画出：

(1) 模块的表示：在结构图中，以矩形框表示模块。在矩形框内注有按功能定义的模块名，也可以简要指明模块的功能或功能名。

(2) 模块间的调用关系：在程序中处理模块间调用关系的原则是：上层模块（又称调用模块）可以调用下层模块（又称被调用模块），而下层模块不能调用上层模块。即第 0 层（也称主模块）可以调用第一层，第一层可以调用第二层，而不能以相反的顺序调用。这种调用方式，可以使模块间的界面清晰、关系简化。

(3) 模块间的通信：在模块和被调用模块间总会有信息传递。图 4.14 中带圆圈的箭头表明了求 C_m^n 的程序模块间的数据传输的内容和方向。

结构图是一个系统开发中的重要资料。结构图的导出，是在对系统详细分析后的产物，目前已有一些比较成熟的方法如结构化设计(SD)方法、Jackson 方法、Parnas 方法等。有兴趣的读者可参阅软件工程方面的书籍。

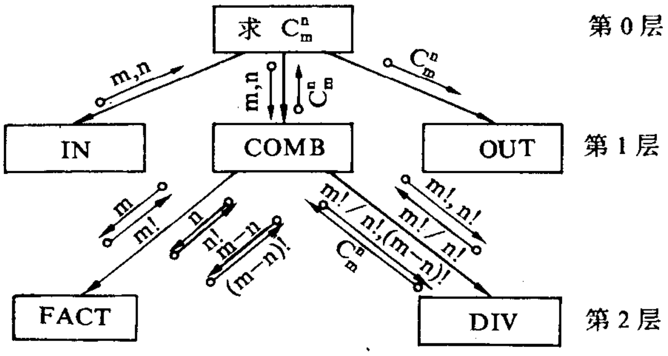


图 4.14

4.4.3 采用自顶向下、逐步细化的设计方法

自顶向下(top—down)方法主要有两层含义:

(1) 在一个复杂系统的设计中,将一个大系统分解为若干个功能模块。采用这种方法进行系统结构设计时,首先要纵观全局,进行总的决策,用一个模块概括(抽象)系统总的功能,形成系统结构图的顶层模块(也称为主控模块或主模块)。然后用若干个下一层模块去表达它,使得总的功能被分解成较为精确的若干个子功能。如此不断逐层向下扩展,直到得到的每一个功能模块都是实现一个基本的功能为止。例如对于求 C_m^n 的问题,首先把问题概括成"求 C_m^n",得到顶层模块。而求 C_m^n 可以用三个子问题:"输入 m 与 n","计算 C_m^n","输出 C_m^n"来表达,形成系统的第一层。其中"计算 C_m^n",又可用"求阶乘"和"求商"两个子问题来表达,形成系统的第二层。不能再分为更小的模块了,一个模块可以用一个函数或子程序来实现。

上层模块的功能是控制或调用其从属(下层)模块。而从属模块是其上层模块功能的具体描述。结构图只反映层次结构,即上下层的调用关系,而不反映执行顺序。假如,一个模块究竟执行多少次,在图上是反映不出来的。

(2) 根据每一个功能模块的要求,在构造实现该功能的算法时,采取自顶而下的设计方法。即根据一个子程序或函数的功能要求,设计出实现它的细节,直到能编写出程序语句为止。算法是反映执行顺序的。在一个大的系统中,程序设计工作是从上层模块入手,还是从下层模块入手呢?在把一个问题表示成层次结构后,最底层的模块便是最具体的。因此,早期的程序设计中,程序员首先把眼光盯在最底层的模块上,先写低层模块的程序。设计好最低层后,再设计它们的上层模块,逐层向上,直到设计完顶层为止。这种实现方法称为自底向上(bottom—up)的方法。如对求 C_m^n 的问题,先设计模块 FACT 和 DIV,再设计 COMB,IN 和 OUT,最后设计顶层模块。

一般说来,自底向上的编程方法,设计出的程序的效率比较高。但是这种方法一开始就把精力集中到程序的细节,而忽视了程序的总体结构,常常出现程序的各部分都设计好了,却因接口关系混乱装配不起来,导致返工甚至整个设计失败。

自顶向下的实现方法要求尽量先安排上层模块的实现,逐步向下实现最基本的功能。即首先安排整个系统的结构及对下层各模块的接口要求,再去实现下层模块。在实现上层模块时,下层模块只作为"树桩"(Stub)出现,即只保留名字、输入参数和输出参数,而本身的处理先空着,或者只保留一两处象征性的语句或简单的显示信息,以供自顶向下地调试时使用。

这样，就能保证整个系统结构的正确性。如果有错误，只是局部的，不会波及整体，导致返工。

当然，在整个自顶向下地实现过程中，有时免不了要出现"回溯"，即在进行下一层的设计时发现上一层不合理，就应返回去对原已确定的某些模块作必要的修改。在这种情况下，也可以采用自底向上的方法，对原先的设想进行改错、补充、加工或优化，并考虑最终求解后实现的容易性。更确切地讲，逐步细化的方法应理解为一种不断地由底向上修正、补充的自顶向下的设计方法。

需要指出：自顶向下、逐步细化与模块化设计的优越性，随着程序的规模增大才愈明显。本书所举的例题，规模都不可能太大，只是为了说明如何使用这种方法。

例 4.4.1 输入 N 个数，然后由键盘控制选做下列三种运算或退出该程序。要求：

键入"1"：输出 N 个数的总和；

键入"2"：输出 N 个数的平均值；

键入"3"：输出 N 个数由小到大的排序结果；

键入"4"：退出。

(1) 确定系统结构：把该程序称为统计程序。可以先把它抽象为图 4.15 所示层次结果。由于"输出结果"是每种计算共有的并且很简单，可以将它并入每个计算中，得图 4.16 所示结构图。

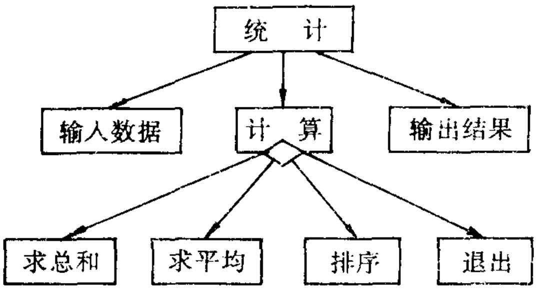


图 4.15

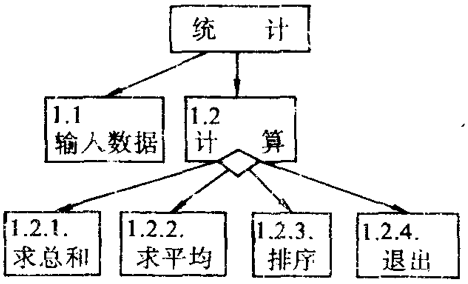


图 4.16

(2) 设计各模块的算法：算法设计自顶向下地进行。通常上层模块是对下层模块的调用、控制和功能分配。在本题中，主模块的功能是定义主要数据结构，对第一层的模块进行调度。由于要顺序接收 N 个数据，所以考虑用一维数组存储它们比较方便。第一层的两个模块是"输入数据"与"计算"，并且是先输入数据，然后再对它们进行计算，所以是顺序结构，其算法如图 4.17 所示。其中双边框表示是一个模块。

下面设计图 4.16 中第一层的两个模块1.1及 1.2 的算法。模块 1.1 的功能是接收输入的 N 个数据，由于采用了一维数组，算法只需一条语句 MAT INPUT 便可以完成。把它命名为"Get—List"。

模块 1.2 的功能是根据键盘上敲入的数选择不同的计算功能，其算法如图 4.18 所示。把它定义为"Count"。

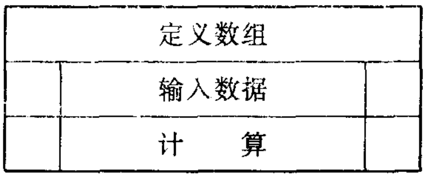


图 4.17

模块 1.2.1 的功能是求 N 个数据的总和。为此必须先测定元素数 N。有了元素数 N 后，便可以用循环结构求和，设数据存在数组D(1：N)中，总和用变量 SUM 存储，可以得到图

4.19 所示算法。把它定义为“Sum”。

1.2

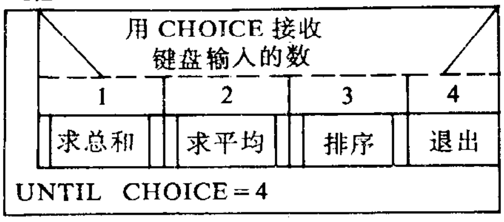


图 4.18

1.2.1

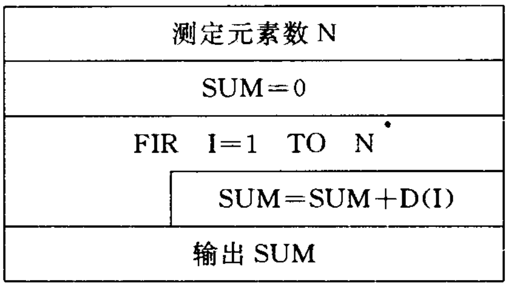


图 4.19

模块 1.2.2 是求平均数。在模块 1.2.1 的基础上，很容易得出图 4.20 所示。求平均数的算法。把它定义为“Auerage”。

1.2.2

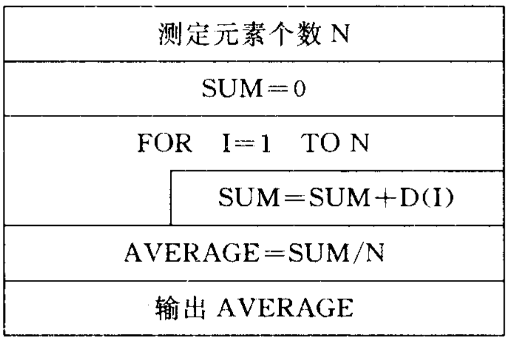


图 4.20

1.2.3

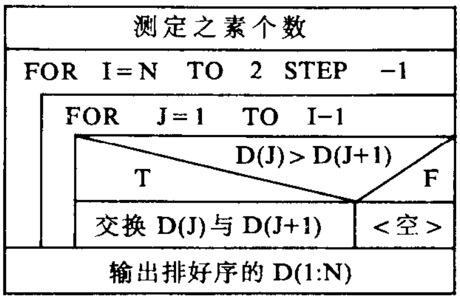


图 4.21

模块 1.2.3 是对 N 个数进行排序。排序算法很多，选用在 3.3 节中介绍过的起泡排序法。在本题中起泡排序法用图 4.21 所示算法描述。把它定义为“Sort—List”。

(3) 编码：对图 4.17～图 4.21 所示算法用 True BASIC 编码，便可以得到 True BASIC 程序。

```
! Program of statistics.
!
dim D(1)                              ! We'll redim later
call Get_list (D)                     ! Get list of items
call Count (D)                        ! Count
end

sub Get_list(D())
  print "Enter items please!"
  mat input D(?)                      ! Redimension to input
end sub

sub Count (D())
  input prompt "Enter your choice:": Choice
  select case Choice
```

```
    case 1
      call Sum(D)
    case 2
      call Average (D)
    case 3
      call Sort_list(D)
  end select
end sub

sub Sum(D())
  let N=Ubound(D)                    ! How many we have
  let S=0
  for I=1 to N
    let S=S+D(I)
  next I
  print "The sum is:": S
end sub

sub Average (D())
  let N=Ubound(D)
  let Sum=0
  for I=1 to N
    let Sum=Sum+D(I)
  next I
let Aver=sum/N
print "The average is:": Aver
end sub

sub Sort_list(D())
  let N=Ubound(D)
  for I=N to 2 step-1
    for J=1 to I-1
      if D(J)>D(J+1)then call Swap(D,J)
    next J
  next I
mat print D;
end sub

sub Swap(D(),J)
  let Temp=D(J)
  let D(J)=D(J+1)
  let D(J+1)=Temp
end sub
```

```
Ok. run
Enter items please!
? 6,7,5,3,8,2,9,1,4↙
Enter your choice: 3↙
  1 2 3 4 5 6 7 8 9
Ok. run
Enter items please!
? 6,7,5,3,8,2,9,1,4↙
Enter your choice: 2↙
The average is: 5
Ok . run
Enter items please!
? 6,7,5,3,8,2,9,1,4↙
Enter your choice: 1↙
The sum is: 45
Ok.
```

4.5 程序的连接

由于机器的内存空间很有限，当 True Basic 程序较大，内存容纳不下时，就无法将程序一次调入内存运行。这时可将源程序分为几段分别存储在外存储器上。True Basic 提供了连接语句，在运行时可以将多个程序连结起来。方法是在前一个程序的末尾加入 Chain（连接）语句，在后一个程序的开头加一条 program（响应）语句。在运行时，先将第一个程序装入内存运行，当运行到 Chain 语句时，就将其后继程序自动地装入内存并运行，直到被连接的最后一个程序运行结束。

4.5.1 连接语句（Chain 语句）

Chain 语句的格式为：

Chain "〈文件名〉" [with (〈串变量〉)] [,return]

使用 Chain 语句应该注意以下几点：

（1）〈文件名〉是被连接程序的文件名。当前程序与被连结程序之间不存在公共变量，每一个程序都是独立的程序单位。

（2）可以通过参数（可以是串变量，也可以是串表达式）向被连结程序传送数据。

（3）当没有 return 选择项时，当前程序所用的内存空间全被释放，除显示器、键盘通道外，其余所有的通道均被关闭，全部变量均被释放。

（4）当有 return 选择项时，保留当前程序及其使用的变量，并且不关闭在用通道。在执行完被连接程序后，仍返回到原来程序，执行 Chain 语句的后继语句。这种带 return 的连接语句类似 Call 语句。已失去了节省内存的优点，只是把程序分解为一些小的程序单位。

4.5.2 响应语句(Program 语句)

Program 语句的格式为：

Program 〈文件名〉[(〈串变量〉)]

使用 Program 语句时应注意以下几点：

(1) Program 语句必须位于被连接程序的开头。如果前一程序的 Chain 语句中无串变量，则 Program 语句可以省略。

(2) 带有 Program 的程序可以单独运行，而带有 Chain 语句的程序不能离开被连接程序单独运行。

例 4.5.1 一个使用 Chain 语句及 Program 语句的例子。

```
! Program name is P1
Print "Program 1"
Chain "P2"
end
! Program name is P2
Print "program 2"
let a$="+++++"
Chain "P3" with (a$)
end
! Program name is P3
Program P3(b$)
Print "Program 3"
print b$
end
```

将程序 P1 调入内存并运行，结果为：

```
Ok. run ↙
program 1
program 2
program 3
+++++
```

如果在两个程序之间需传送数值型的量，在连接程序中可将数值型的量转换为字符型，再做为参数传递给被连接程序，被连接程序接收到参数后再转换成数值型。

如果连接程序中有多个量需被传送，则可以将多个量都转换成字符型，再按一定顺序连接成一个字符串，并且中间以特殊符号隔离。在被连接程序中再将串变量分解。

习 题

4.1.1 写出计算下列各式的函数定义：

(1) $y=\dfrac{\ln(1+\sqrt{x})}{1+x^2}$;

(2) $Z=\dfrac{x^2+y^2+a}{(x^2+y^2)^{2/3}}\cdot \mathrm{tg}(x^2+y^2)^3$;

(3) $W=\dfrac{1-a}{1+a+a^2}\left(\dfrac{1-b}{1+b+b^2}+\left(\dfrac{1-c}{1+c+c^2}\right)^2\right)$;

(4) $u=\dfrac{1+\sin^2x}{\sin^2(wy)+\sin^2(Z+\beta)}$;

(5) $Y=1+\dfrac{x^2}{2!}+\dfrac{x^4}{4!}+\dfrac{x^6}{6!}+\dfrac{x^8}{8!}$;

(6) $W=\dfrac{f(x)+f(y^2)}{f(x+1)+y^2}\cdot f(f(t))$

其中 $f(u)=u-\dfrac{u^2}{6}+\dfrac{u^5}{120}$

4.1.2 指出并分析下列程序中的错误：

(1)
```
def x(x,y$)
  for i=1 to x
    let y$=y$+"*";
  next i
end def
print x(2,"x")
end
```

(2)
```
def x(a)
  if a<0 then
    let x=1
  else if a=0 then
    let x=x*x
  lest
    let x=2*x
end if
input x
print x(x)
end
```

(3)
```
mat print r;
dim r(10), a(10)
def r
  mat input a(?)
  mat r=a
end def
end
```

(4)
```
input n
print #1: n
end
def a(#999,n)
  if n=1 then
    exit def
  else
    open #999:printer
  end if
end def
```

4.1.3 求 7832，534，15842，2492 的最大公约数与最小公倍数。

4.1.4 计算 n 维空间中点$(x_1,x_2,\cdots,x_n)$和点$(y_1,y_2,\cdots,y_n)$之间的距离。

$$r(x,y)=\sqrt{\sum_{i=1}^{n}(x_i-y_i)^2}$$

4.1.5 判断一个正整数是否质数。如果是，则显示出“yes”；不是，则显示出“NO.”

4.1.6 从一个 M×N 数组的某个指定行内，找出绝对值最大的元素所在的列号。

4.1.7 用递归函数求 Fibonacci 数列。

4.1.8 用递归函数计算 $\sum_{i=1}^{N} i$。

4.1.9 Ackermann 函数定义在非负整数 n，x，y 上

$$ACK(n,x,y)=\begin{cases}x+1 & (n=0)\\ x & (n=1,y=0)\\ 0 & (n=2,y=0)\\ 1 & (n=3,y=0)\\ 2 & (n\geqslant 4,y=0)\\ ACK(n-1,ACK(n,x,y-1),x) & (n\neq 0,y\neq 0)\end{cases}$$

它是一个多重递归函数,如计算 ACK(2,3,1)的过程如下:

ACK(2,3,1)=ACK(1,ACK(2,3,0),3)=ACK(1,0,3)
=ACK(0,ACK(1,0,2),0)
=ACK(0,ACK(0,ACK(1,0,1),0),0)
=ACK(0,ACK(0,ACK(0,ACK(1,0,0),0),0),0)
=ACK(0,ACK(0,ACK(0,0,0),0),0)
=ACK(0,ACK(0,1,0),0)=ACK(0,2,0)=3

试编一计算 Ackermann 函数值的程序。

4.2.1 指出下列程序中的错误:

(1)
```
sub x$(x,y,z$)
  let x=""
  for i=x to y
    let x$=z$+z$
  next i
end sub
input a$,b,c
call x$(a$,b,c)
print z$
end
```

(2)
```
data 1,2,3,4,5,6,7,8
call read (a)
mat print a;
end
sub read (r())
  for i=1 to 8
    read r(i)
  next i
end sub
```

(3)
```
  input a,n
  for i=1 to n
    call a;
  next i
  sub divide
    print a
  end sub
end
```

(4)
```
dim a(1)
input a(?)
call print
end
sub print
  mat print a
end sub
```

4.2.2 把数组 A(200)中的前 N 个元素颠倒排列。

4.2.3 输入一个 M×N 的字符数组(每个元素为一个字符),打印出把它顺时针转 90°后的数组。

4.2.4 产生一个 10×10 的二位随机整数方阵,试编程序完成下述任务:先求出每一行的最大值和每一列的最小值;然后求 10 个最大值中的最小者,10 个最小值中的最大值;最后求这两个数之差的平方。

4.2.5 编程输出如图 4.22 所示的数字旋转方阵,这是一个 6×6 的方阵,希望能编出 n×n 的数字方阵4≤n≤10。

1	20	19	18	17	16
2	21	32	31	30	15
3	22	33	36	29	14
4	23	34	35	28	13
5	24	25	26	27	12
6	7	8	9	10	11

图 4.22

4.2.6 用子程序求 N 个正整数的最大公约数和最小公倍数。

4.2.7 编制一个程序,输入任意一个日期(年、

月、日)后,能输出它是星期几。

4.2.8 编制一个打印N阶魔方阵的程序。魔方阵的元素为$1\sim N^2$的自然数,方阵的每一行,每一列以及对角线的元素之和统统相等。

提示:魔方阵元素的排列有以下规律:

① 自然数1总是在方阵第1行的当中一列上;

② 后继的自然数在前一个自然数的右上方,即行数减1,列数加1。当行数为1时,下一个自然数将在第N行;当列数为N时,下一个自然数将在第1列;

③ 如果该放下一个自然数的位置已被占据,则下一个自然数应放在前一个自然数的下面,即行数加,列数不变。

4.2.9 设有若干根火柴,分作数堆,堆数和每堆的火柴根数都是任意的。试编制一个人机游戏程序,该游戏的规则是:

① 人机双方轮流拿火柴;

② 每次拿取至少一根,且只许从一堆中拿取;

③ 拿到最后一根者获胜。

4.2.10 用递归子程序求Fibonacci数列。

4.2.11 用递归子程序求N个数的最大公约数和最小公倍数。

4.2.12 八皇后问题:国际象棋中皇后的威力最大。它可以在横、竖及对角线方向上不受距离限制地移动吃子。在8×8的国际象棋棋盘上,如何安放八个皇后,使她们互不吃掉?

4.2.13 函数与子程序有哪些异同?

4.3.1 一个有主程序的程序是否可作为库文件保存?

4.4.1 有人说,我的脑子特别好,即使程序不分块,我也能完成它。假如他的话是真的,是否允许他这样做?为什么?

4.4.2 是不是程序模块划分得越小越好?

4.4.3 划分模块的依据是什么?

4.4.4 把模块组织成层次结构有什么好处?

第五章　字　符　串

字符串是一类重要的数据。True BASIC 具有很强的字符串处理功能，可以适应字符处理的多种需要。在第二章中，已介绍过字符串的初步概念、字符串连结及子串赋值。这一章将较全面地介绍 True BASIC 的字符串处理功能。

5.1　基本概念

5.1.1　字符串常数与字符串变量

字符串常数也称字符串文字或字符串，通常是由包含在两个双引号之间的若干个字符组成。下面是几个字符串常数的例子：

"GW0520 MICRO-COMPUTER"(含有空格的字符串)

" "(一个空字符串)

"He Laughed" "Ha ha" "and fell backward"(含有双引号的字符串)

字符串常数中的字符是用户根据需要设定的。应当注意：

(1) True BASIC 虽然对符号名(变量名、子程序名、函数名等)是不区分大小写的，但字符串中的大小写字母是有区别的。如“job”与“Job”被看成是两个不同的字符串。

(2) 空格是一种特殊的字符。如“␣␣ A”与“␣ A”被看成是两个不同的字符串。

(3) 双引号也可以包含在一个字符串常量内，但必须重复引用两次。即在字符串中要用两个连续的引号代表一个引号。如"X" "Y"表示由三个字符——“X”、引号和“Y”组成的字符串。而" "　" "表示只含一个引号的字符串。

```
! Ask for string and number,
! and print it
!
input string $ , num
print string $ ; num
end
```

```
Ok. run
? "Today's" "number" "is:",↙
? -2.5e2↙
Today's "number" is: -250
Ok.
```

但是，如果由键盘键入

"Today's" number "is:"

则会出现 Illegal data 的错误，告诉你所键入的是非法数据。

在程序中，一个字符串常量可以存储在一个字符串变量中，也可以把若干字符串按一定的顺序存储在一个字符串数组中。

5.1.2 字符串的比较

在 2.5.1 节中，已经介绍了字符串比较的基本方法。对一组字符串排序，就是通过一定规则的比较才能实现。

例 5.1.1 有一批国家名，找出按英文字母顺序应排在最前面的国名。

这个程序比较简单，将这些国家名存放在字符串数组 a$ 中，将第一个国家名(存放在 a$(1)中)依次与后面的各个国家名(存放在 a$(2)到 a$(n)中)分别比较，每次把比较中较小者与 a$(1)交换即可。

程序如下：

```
dim a$ (5)
mat read a$
let u=Ubound(a$)
for i=2 to u
  if a$ (1)>a$(i) then call swap
next i
print a$ (1)
data "China","Japan","Canada","Korea"
data "England","France","America","India"
sub swap
  let x$ =a$(1)
  let a$(1)=a$(i)
  let a$(i)=x$
end sub
end
--------------------------------------------------
Ok. run
Canada
Ok.
```

5.1.3 字符串连接与子字符串

字符串连接与求子字符串是字符串的两种基本运算。字符串连接要使用连接符“&”。子字符串(也称部分字符串)则用它在整串中占的位置来表示：

〈字符串〉[〈开始位置〉:〈结束位置〉]

例如，有一个字符变量 S$，其值为"The Peaple's Republic of China"。如果有 S$(26:30)，则此子串的值为“China”。

这两种运算已在 2.3 节中介绍，这里不再赘述。

5.2 字符串传送

同传送数值数据一样，用 LET 语句，READ 语句及 INPUT 语句可以分别从运算器、程序数据区及键盘把字符串常量传送到内存变量中去，也可以用 PRINT 语句从运算器中把字符串表达式的值传送到输出设备——显示器或打印机。这些传送方式也已在 2.4 节中作过介绍，下面仅介绍传送字符串时一些具有特殊性的问题。

5.2.1 用 READ/DATA 或 INPUT 语句传送字符串常量

READ/DATA 或 INPUT 语句可以从程序数据区或键盘把一组常数传送到对应的内存变量中去。当要传送字符串常量到相对应的字符串变量中去时，该字符串常量可以用双引号括起来，也可以省略双引号。有几点注意：

(1) 字符串常量不用引号括起来时，如果字符串中有首尾空格，在传送到字符串变量中时，其首尾空格被删去。

例 5.2.1

```
input a$,b$
print a$;b$
end
```

```
Ok. run
? abcd ␣␣␣,↙
? ␣␣␣ dcba↙
abcddcba
Ok.
```

因此向字符串变量传送具有首尾空格的字符串常量(包括传送一个空串)时，必须用双引号将字符括起来。

(2) 在 True BASIC 中逗号是用来分隔两个数的，双引号是字符串的起止符号。因此如果要传送的字符串中带有逗号或双引号时，必须在字符串两端加双引号。

例 5.2.2

```
for i=1 to 2
  read a$,b$
  print a$;b$
next i
data Oct. 1,1988,Beijing
data "Oct. 1,1988",Tai yuan
end
```

```
Ok. run
Oct. 11988
BeijingOct. 1,1988
```

```
Ok
```

在第一次执行 read 语句时，将“Oct. 1”作为第一个字符串常量传送给 a $，将“1988”传送给 b $。在第二次执行 read 语句时，将“Beijing”传送给 a $，把“Oct. 1，1988”传送给 b $。

例 5.2.3

```
  read a$ ,b$
  print a$ ;b$
data Today's "number" is, -250
end
```

```
Ok. run
Illegal data.
Ok.
```

DATA 语句中的字符串要改为"Today's" "number" "is:"才行。

(3) 惊叹号是 True BASIC 规定的注释标识符。因此，DATA 语句中的字符串常量中含有惊叹号时，必须将该字符串常量用双引号括起来，否则 True BASIC 将把惊叹号后面的部分当作为注释。

例 5.2.4

```
read a$ ,b$
data Look out!, this is a data
print a$ ,b$
end
```

```
Ok. run
Reading past end of data.
Ok.
```

DATA 语句中的惊叹号，使 True BASIC 把 DATA 语句看作只有一个数据“look out”，当 b $ 读数据时，数据区指针已超出数据区，所以出现“Reading past end of data”的错误信息。

5.2.2 LINE INPUT 语句

LINE INPUT 语句能够输入完整的一行字符串，该字符串中可以包含任何字符——包括逗号、引号或前后空格。

使用 LINE INPUT 语句时，由键盘输入的一行字符串将被逐个读入，并传送到一个字符串变量中去，不论输入的是什么字符，都能正确地输入。

```
! Read a book list and print it again.
!
do
  line input prompt "Book?":book$
  if book$ ="" then stop
```

```
  print book $
  print
loop
end
```

```
Ok. run
Book? Tan, "BASIC Language"↙
Tan, "BASIC Language"
Book? Zhang, "programming in BASIC"↙
Zhang, "programming in BASIC"
Book? ↙
Ok.
```

LINE INPUT 语句的一般形式是

LINE INPUT〈变量 1〉,〈变量 2〉,……

LINE INPUT PROMPT〈提示字符串〉:〈变量 1〉,〈变量 2〉,……

MAT LINE INPUT〈数组 1〉,〈数组 2〉,……

MAT LINE INPUT PROMPT〈提示字符串〉:〈数组 1〉,〈数组 2〉,……

这里每一个变量或字符串数组元素都需要单独输入一行字符串。

5.2.3 用 LET 语句传送字符串数据

用 LET 语句可以把一个数值表达式或字符表达式的值传送给一个或多个变量。这种传送称给变量赋值。用 LET 语句给字符串变量赋值时应注意以下两点:

(1) 在赋值号右边的字符串表达式中,若有字符串常量,则必须把它用双引号括起来。如果有人写出以下语句

LET month $ =January

在执行时,True BASIC 认为赋值号右边的 January 是一个数值变量名,而赋值号左边却是字符串变量,于是给出"Wrong type"(类型错)错误信息。

(2) LET 语句还可以给一个字符串的子串赋值,即将一个字符串传送到另一个字符串的一部分中去。

例 5.2.5

```
let a $ ="ABCD"
let a $ [2 : maxnum]="bove"
print a $
end
```

```
Ok. run
A bove
Ok.
```

将字符串变量 a $ 中第二个字符到最后一个字符，用“bove”替换。但是上述语句不能写成

let "ABCD"[2：maxnum]="bove"

因为变量中的内容是可以改变的，而常量的值是不可以改变的。赋值号左端必须是变量，不能是常量。否则会出现“Wrony type”的错误信息。

5.3 字符串函数

字符串函数可以实现特定的字符串操作。True BASIC 提供有下列五类字符串函数：

(1) 测字符串长度函数；

(2) 字符串转换函数；

(3) 字符串重复函数；

(4) 字符串查找函数；

(5) 删除首尾空格函数；

这些函数或者得到(或称“返回”)字符串值；或者以字符串为自变量；或者两者兼有之。下面分别介绍这些函数。在这些函数的说明中，为叙述方便起见，除特殊注明外，a $ 和 b $ 都代表字符串表达式，n 代表数字表达式。

5.3.1 测字符串长度函数

函数 LEN(a $)可以返回一个字符串 a $ 的长度。即字符串 a $ 中所含字符的个数。如 LEN(“three”)返回 5，LEN(“True ␣ BASIC ␣ Language”)返回 19，LEN(" ")返回 0。

例 5.3.1 从一个给定的字符中删去某些字符，如原有字符为“FORTRAN 77 LANGUAGE”，删去“A”和“R”则变成“FOTN 77 LNGUGE”。算法如图 5.1 所示。

程序及运行结果如下：

```
print "Input a phrase:"
input p $
print "Input string to be crunched out:"
input k $
let answer $ =" "
for p=1 to len (p $ )
  let flag,k=1
  do while k<=len (k $ )and flag=1
    if p $ [P : P]=k $ [k : k]then let flag=0
    let k=k+1
  loop
  if flag=1 then let answer  $ =answer $  & p $ [p : p]
next p
print answer $
end
```

```
Ok. run
Input a phrase:
? RAILROAD CROSSING WITHOUT ANY VOWEILS↙
Input string to be crunched out:
? AEIOUY↙
RLRD CRSSNG WTHT N VWLS
Ok.
```

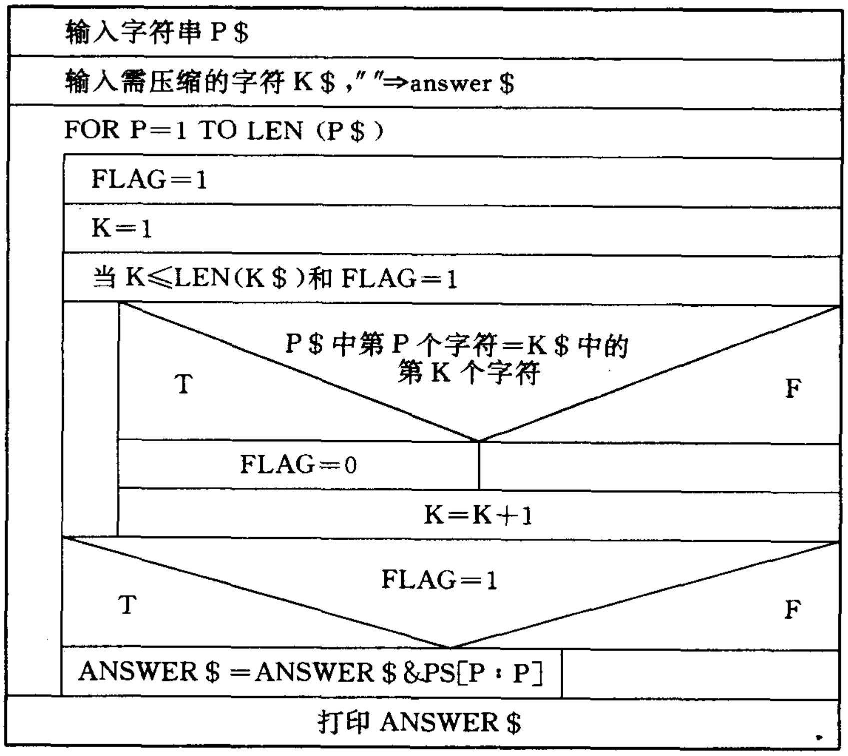


图 5.1

5.3.2 字符串转换函数

True BASIC 的字符串转换函数有如下几类:

(1) 大、小写转换函数;

(2) 数字字符串与数值数据的转换函数;

(3) 字符与 ASCII 码的转换函数;

下面分别介绍这几类函数。

1. 大小写转换函数:函数 LCASE $ (a $)能够把 a $ 中的大写字母改写成小写字母,其它字母不变。如表 5.1。

函数 UCASE $ (a $)则可以把 a $ 中的小写字母改写为大写字母,其它字母不变。如表 5.2。

这两个函数多用于处理用户输入的数据,不论用户输入的数据是大写还是小写字母,程序都可以接受,并有效地处理。

表 5.1

调用	返回
Lcase $ ("Wyatt Earp")	"wyatt earp"
Lcase $ ("MaRainey")	"marainey"

表 5.2

调用	返回
Ucase $ ("The First State")	"THE FIRST STATE"
Ucase $ ("Price= $ 10.95")	"PRICE= $ 10.95"

例 5.3.2

```
do
  print "Command";
  input cmd $ ,
  let cmd $ =Lcase $ (cmd $ )
  if cmd $ ="stop" then stop
  input string $
  select case cmd $
    case "ucase"
      print Ucase $ (string $ )
    case "lcase"
      print Lcase $ (string $ )
    case else
      print "Try Ucase, Lcase, or stop"
    end select
  loop
  end
```

```
Ok. run
Command? lcase, HAPPY BIRTHDAY↙
happy birthday
Command? ucase, Many Happy Returns↙
MANY HAPPY RETURNS
Command? stop↙
Ok.
```

2. 数字字符串与数值数据的转换函数:函数 STR $ (n)可以把数值数据 n 转换为一个由数字组成的字符串,如右表。

函数调用	返回
Str $ (3+.14)	"3.14"
Str $ (.0000000009)	"9.e—10"
Str $ (2—5)	"—3"

调用 STR $ 函数时先算出括号内的值,然后转换成数字串形式。其形式与 PRINT 语句所应显示的格式一样,只是没有加上尾随空格和正数前的空格。例如,打印数值0.0000000009的形式为 9.e—10,因此 Str $ (0.0000000009)的值为"9.e—10"(无尾随空格和正数的前导空格)。

例 5.3.3

```
let a=3
```

```
let b=.14
print a+b
print str$(a+b)
end
```

```
Ok. run
␣3.14(数值)
3.14(字符串)
Ok.
```

输出的结果中，一个是数值数据，前面要空一个符号位，另一个是字符数据，前面不空符号位。

函数 VAL 是函数 STR $ 的逆运算。它能将一个由数字组成的字符串，转换成一个数值数据。在转换过程中，字符串的首尾空格将被忽略，并且要求这个串能表示成正确的数值，否则就给出出错信息。如表 5.3 所示。

表 5.3

函数调用	返　回
Val("3.1415")	3.1415
Val("1.945e+3")	1495
Val("1.23.45")	"VAL String isn't a Proper number"(出错)

例 5.3.4　一种用复验法提高输入数据的可靠性的方法

对重要的数据，为保证其正确性，往往采取两次输入的方法。若两次输入的值相同，就认为输入的数据正确，否则就可能是错误的，要求你重新输入。下面的程序便是其中一种方法。

```
clear
input prompt    "请输入数据:": D$
input prompt    "请再输入一遍:": D$
do while D$< >D1$
  input prompt "错误,重新输入": D$
  input prompt "请再输入一遍:": D1$
loop until D$=D1$
let D=val(D$)
⋮
      (后继语句)
⋮
```

本来程序希望输入一个数值量给 D，而程序中却用了字符串变量 D$，D1$ 接收输入数据，这是为了避免在输入时误按了某个非数字字符键，造成"数据与变量类型不同"的错误而使程序停止运行。现在用字符串变量，即使输入了非数字字符也不会使程序停止运行。若检查出两次输入的字符串不相同，就认为输入了不正确的数据，要求用户重新输入数据(此时程序不会停止运行)。如果检验了两次输入相同则认为输入正确。然后再把它转换为数值量，

赋值给 D。当然，也可能发生二次输入相同而都是错误输入的情况(例如看错了数据)。因此，这种方法并不是绝对可靠的，而只是相对地提高了可靠度。

除 STR $ 与 VAL 外，True BASIC 还提供了另外一对函数 NUM $ (n)及 NUM(a $)。NUM $ 函数能把一个数转换为用 8 个字节表示的数字串。在 8 个字节(8×6=64 位)中，符号占一位，指数 11 位，数字部分 52 位。注意它不是数值串，而是字符串形式，这些特殊形式的字符串不能在屏幕上正确地显示，但可以把它们输出到磁盘文件中去。NUM $ 常用作把一个数值转换为字符串，然后把它们作为一个记录送到磁盘上的记录文件中。

NUM 是 NUM $ 的逆运算。

3. 字符与 ASCII 码的转换函数：函数 ORD 可以返回 ASCII 码表中任何一个字符或控制字符所对应的 ASCII 码值。(参考附录 D)，如表 5.4 所示。

表中"KA"是 True BASIC 不能识别的一个 ASCII 码，例如有的计算机系统使用的特殊字符，所以返回错误信息。

函数 CHR $ 是 ORD 的逆运算。它以一个数值作自变量，返回这个数值所对应的 ASCII 码表中的一个字符。字符代码 0～127，所返回字符是标准的，128～255 在不同的机器上可能有所不同。IBM PC 上所用的字符集见附录 D。

若所使用的数 n 超出范围 0～255，则 True BASIC 用 MOD(n，256)把 n 折合到这个范围之内。如表 5.5。

表 5.4

调　用	返　回
ord ("A")	65
ord(" * ")	42
ord("　")	−1
ord("␣")	32
ord("KA")	"Improper ORD string"

表 5.5

调　用	返　回
Chr $ (65)	"A"
Chr $ (32)	"␣"
Chr $ (65+256)	"A"
Chr $ (65.2+256)	"A"
Chr $ (65.6+256)	"B"

例 5.3.5 密码的编码/译码程序：为提高数据的安全性，有时需要将数据以密码的形式存放。编制密码就是将一个字符串按一定的规律转换为另一个新的字符串，以达到保密的目的。编码的方法很多，我们采用的方法是：先取每一个字符的 ASCII 码，将此值加 5，再用 CHR $ 函数把它转换为一个新的字符。由于 CHR $ 函数的自变量 n 最大为 255，如果某一字符的 ASCII 码加 5 已超过 255，则取其与 255 的差值进行编码。用这种方法，字符"B"的 ASCII 码为 66，加 5 得到 71，CHR $ (71)为"G"，故将"B"编为密码"G"。译码是编码的反变换。

下面是一个编码/译码程序及运行情况：

```
! Coding
!
print "Enter message:";
input old $
let nu $ =" "
for k=1 to Len(old $ )
```

```
    let tran $ =old $ [k : k]
    let code=Ord(tran $ )+5
    if code>=255 then let code=code-255
    let nu $ =nu $ & Chr $ (code)
next k
print "The coded message is:"; nu $
! Decoding
let old $ =" "
for k=1 to Len(nu $ )
    let tra $ =nu $ [k : k]
    let code=Ord(tra $ )-5
    if code<=0 then let code=code+255
    let old $ =old $ & Chr $ (code)
next k
print "The decoded message is:"; old $
end
```

```
Ok. run
Enter message:? BASIC↙
The coded message is: GFXNH
The decoded message is: BASIC
Ok.
```

5.3.3 字符串重复函数

函数 Repeat $(a$,n)以 a$与 n 为自变量,并得到 n 个连续的 a$。见表 5.6。

5.3.4 子字符串查找函数

函数 POS(a$,b$,n)的作用是在字符串 a$中第 n 个字符位置开始找字符串 b$,得到的函数值就是 b$第一个字符的位置。若 n 小于等于 1,则 True BASIC 用 1 代替,在这种情况下,n 可以省略,即可以写成 Pos (a$,b$)的形式。若 n 大于字符串 a$的长度,或无匹配串,则 Pos 返回"0"。如表 5.7。

表 5.6

调 用	返 回
Repeat $ ("OX",3)	"OX OX OX"
Repeat $ ("cow",0)	" "
Repeat $ ("X",−1)	出错(Repeat $ connt<0)

表 5.7

调 用	返 回
Pos ("Shang hai", "ha",1)	2
Pos ("Shang hai", "ha",3)	6
Pos ("Shang hai", "han",3)	0
Pos ("Shang hai", "ha",10)	0
Pos ("Shang hai", "ha",)	2

例 5.3.6 查找图书：图书馆中有许多书(如 1000 本)，你想借某本书，却忘记了全名，只记得其中几个字，希望程序能帮你把有这几个字的书名都显示出来。

程序如下：

```
print "Enter your words:";
input words $
do
  read book_name $
  let f=Pos(book_name $ ,words $ )
  if f>=1 then print book_name $
loop until End data
data ALGOL, FORTRAN LANGUAGE, LISP, FORTRAN 77
data PROBLEM SOLVING WITH BASIC, BASIC LANGUAGE
data PROGRAMMING IN TRUE BASIC, COBOL, PASCAL
data PROLOG,…………
end
```

```
Ok. run
Enter your words:? BASIC↙
PROBLEM SOLVING WITH BASIC
BASIC LANGUAGE
PROGRAMMING IN TRUE BASIC
Ok.
```

5.3.5 删除首尾空格函数

函数 LTRIM $ (a $)可以返回删除字符串 a $ 的开头空格后的字符串。函数 RTRIM $ (a $)可以返回删除字符串 a $ 的结尾空格后的字符串。函数 TRIM $ (a $)可以返回删除 a $ 的前后两端的空格后的字符串。

例 5.3.7

```
let a $ ="␣␣Hello! ␣␣"
let b $ ="␣␣Ok! ␣␣"
print Ltrim $ (a $ ); Ltrim $ (b $ )
print Rtrim $ (a $ ); Rtrim $ (b $ )
print Trim $ (a $ ); Trim $ (b $ )
end
```

```
Ok. run
Hello! ␣␣Ok! ␣␣      (删去首空格)
␣␣Hello! ␣␣Ok!      (删去尾空格)
Hello! Ok     (删去首尾空格)
Ok.
```

习　题

5.1.1　在下列◯内填入合适的关系运算符，使关系表达式的值为真。

"CAT"◯"DOG"　　"MAN"◯"MEN"

"IT ␣ S"◯"ITS"　　"X(2)"◯"X2"

" "◯" " " "

5.1.2　指出下列程序执行后的打印结果：

(1)
```
let x$="Go ␣"
let y$="To"
let z$=x$ & y$ & "␣ 100"
let a$=z$[4:6]
print x$; a$&"10"
end
```

(2)
```
let a$="ABCLEFGHIJKLMN"
print (a$[3:Maxnum] & a$[1:5])[5:7]
end
```

5.2.1　指出下列程序段中的错误

(1)
```
read auto $, b, code$, def$
print b; auto$, def$, code$, c
data cil, a, pen, let
  ⋮
  ⋮
```

(2)
```
input prompt "data?":d
print "Input from start";
input d$
if d=d$ then
   print d
end if
  ⋮
  ⋮
```

(3)
```
let a$=Happy
print a$
  ⋮
  ⋮
```

(4)
```
read a$,b,c$,d
print a$,c$,b,d
  ⋮
  ⋮
data ␣␣ Ok! ab,012,3.45,678
```

5.2.2　编一个程序，能把输入的字符串中的“计算机”改为“电脑”。程序中的字符串使用汉语拼音。

5.3.1　设计一个程序，给一批数码前面加“0”，使得它们的位数与这批数码中位数最长的数码的位数相同。

5.3.2　设计一个程序，当输入一段英文后，能统计出其中有多少个元音字母。

5.3.3　设计一个程序，不论输入的文字都是大写，还是小写，或大小写任意混合，都能将其整理成除每个句子开头字母是大写外其它都是小写的文字。

5.3.4　修改例 5.3.6 中的程序，使得书名及关键词不管是大小写均能正确查出有关书名。

第六章　程序设计方法和风格

6.1　程序质量的标准

6.1.1　关于程序质量的几个概念

1. 正确性：当给出正确的输入数据时，程序能给出正确的结果，就说程序是正确的。正确性是程序的最根本的质量标准。

2. 可靠性：可靠性是比正确性更广泛、更深刻的一个概念。当设计好一个程序后，总要先用几个数据测试一下该程序是否正确。但是，常常有这样的情况：用某些数据进行测试时，能得到正确的结果，而用另外一些数据进行测试时，会得出错误的结果。也就是说，如果使用的数据的“覆盖面”小，就有可能把一个有错误的程序误认为一个完全正确的程序。

例 6.1.1　设计一个判断某个数是否素数的程序。是，则输出“yes”，不是，则输出“No”。有人编出的程序如下：

```
input n
if n/2=Int (n/2) then
  print "No."
else
  print "yes."
end if
end
```

当输入是 3,4,5,6,7,8,10,11,12,13,14,17,18,19,……时，结果是正确的。然而在其它情况下，如输入的是 9,15,21 等时，便是不正确的。原因是这个程序只能判奇偶数，对奇数打印“yes”。而不能辨别奇数是否素数。

提高程序正确性的另一有效手段，是采用合适的程序测试技术。程序测试的关键是设计测试用例，即测试用数据集。但是随着程序的增大，对它进行全面测试是一件很复杂，并要付出很大代价的工作。通常应在可靠性与代价之间进行权衡。

除了正确性之外，在更广泛的意义下，可靠性还包括程序的健壮性。

例 6.1.2　测试一个数是否素数的程序：

```
input n
for i=2 to Int (Sqr (n))
  if n/i=Int (n/I) then
    print "No"
    exit for
end if
next i
```

```
if i>Int (Sqr(n)) then print "yes"
end
```

这个程序比例 6.1.1 的正确性提高了。但是当输入 1 时，却打印“yes”，而 1 不是素数，当 n 为其它自然数时，程序才是正确的。再看另外一个例子。

例 6.1.3 求商的程序：

```
input m, n
let q=m/n
print q
end
```

这个程序很简单，乍看起来也没有什么问题。但是，当给出输入数据 N=0 时，程序将因出现“Division by zero.”的错误而非正常地停止运行。

一个健壮的程序在遇到用户有意或无意的不合理输入时，系统应能拒绝这种输入，并指出输入的不合理性，提醒用户注意。如把例 6.1.3 改为：

```
input m, n
if n=0 then
  print "Divisor is zero, redo from start."
let q=m/n
print q
end
```

程序就是健壮的了，也是正确的了。就可以把它称为可靠的了。

一个正确的程序，可以不检查输入数据是否合理（认为输入数据是正确的），而这可能会造成严重后果。而一个可靠的程序则要求在大多数情况下能可靠的工作。

除此之外，健壮性还意味着系统对环境变化的适应性。例如遇存储溢出，硬件故障等意外事件发生时，系统能按照某种预定的方式适当处理，有效地控制事故的蔓延，不致丢失重要的信息。

3. 可用性：程序是供用户使用的。可用性是指用户使用该程序的方便程度。在考虑可用性时，应恪守“与用户友好”的原则，即要考虑用户使用、操作时是否方便、灵活，不易出错。

4. 效率：效率是指程序的运行速度和所占的存储空间大小等。如例 2.6.4 与例 4.2.7 都是求两个数的最大公约数。由于后者使用了递归，每次递归调用都要产生一些局部重复量，因而它所占用的内存存储空间的大小及执行速度都比使用迭代法的前者要差。在 2.7 节中，也曾分析过，多重循环的程序中，当各层循环次数不同时，把重复次数多的循环放在内层与放到外层，程序的效率是不同的。

5. 可维护性：一个程序投入运行以后，由于下列一些原因，还会对其作一些改动：

（1）一般说来，正确性并不是绝对的，对大型程序来说，经过正确性与测试代价之间的平衡，程序仍然会潜伏一些错误，当程序运行以后，某些错误才逐渐暴露出来，这种纠错过程称为校正性维护。

（2）程序投入运行以后，其运行环境可能会发生改变，如硬件技术的进步，新的机型问世，操作系统的更新等。为适应变化了的环境，程序需要进行适应性维护。

(3) 一个程序投入运行后，用户会提出新的要求，这时要对程序进行完善性维护。

(4) 一个程序投入运行后，为了更有生命力，或为将来的改进奠定基础，需对它进行修改或补充。这种维护称为预防性维护。

程序的可维护性在数据处理业务中十分重要。据统计，一个数据处理系统的维护费用可高达开发费用的 3 至 5 倍。

6. 可移植性：一个程序总是在一定的计算机环境下实现的。所谓可移植性是指把在某个计算机环境下的程序，变成为另一个计算机环境下的程序。

7. 可理解性：程序不仅要让机器理解并执行，还要供人阅读。

一个程序员在编写、测试和维护过程中不仅自己要反复地阅读程序，常常还需要别人来阅读和理解。有的程序员编出的程序不仅别人难以看明白，甚至过一段时间后，自己也看不懂了。可想，这样的程序是难以维护的。所以，易理解性是提高可靠性和易维护性的前提。

易理解性也称可阅读性或易读性。它有两重含义：一方面是指程序本身的结构清晰，不仅程序设计者容易阅读和理解，而且它的人机界面清晰，用户易于使用。另一方面，还应为程序编写有关的文档资料，以提高程序的编写、测试及维护的质量和效率，给用户操作提供说明。在计算机系统中，软件的全面的含义应包括以下三部分：① 程序；② 程序运行时所需的数据；③有关的文档资料。有的人不注意组织文档资料，以为设计出一个程序并能正确运行，任务就完成了。这是不对的。应当从软件工程的角度来组织软件生产的过程。养成严密而良好的工作习惯。

6.1.2 从效率第一到清晰第一

前面讨论了程序质量的概念，除了应保证正确性和可靠性外，如何全面考虑各个因素，尤其是如何正确处理效率与可理解性的关系呢？对这个问题的答案是和历史背景相联系的，它依赖于程序设计的环境和人们对程序的基本要求。

在计算机发展的早期，它的内存很小，速度也不高，价格却十分昂贵。人们买到一台计算机如获至宝，总想让它做更多的工作，发挥更大的效益。于是，这项艰巨的任务便落到了程序员肩上。为了在小的可怜的内存的支持下解决大问题，为了节省价格贵得惊人的机时，程序员不得不冥思苦想地构造一些“技巧”，象艺术创造一样编造各种精巧的程序。程序的效率（占用的存储空间和机时的多少）成了评价程序好坏的主要标准，程序员也常以效率当作炫耀自己聪明才智的资本。这样的程序设计环境造就了“效率第一”的程序设计思想。这种程序设计思想给予程序员个人的自由度太大：编制程序无章可循，各行其事；程序产生是一种手工方式，程序的质量完全决定于程序员个人的“艺术”水平；编制出的程序效率很高，可读性却很低，给交流、修改、测试、维护造成很大的障碍。

50 年代中期，晶体管的出现使计算机技术出现了一次转机，机器性能大大提高，价格大幅度下降。60 年代中期，集成电路技术又把计算机带入第三代，大批质优价廉、功能强大的计算机大批涌入市场，高级语言蓬勃兴起，计算机应用迅速地渗透到各学科和技术领域。与此同时，程序的规模也在急剧增大。60 年代末到 70 年代初，软件的规模出现了难以控制的急剧膨胀。这给程序设计带来了新问题：软件规模急剧膨胀，软件的设计方法却仍停留在手工业方式；程序设计的效率很低，许多大型软件要花费几千人的劳动，软件的成本急剧上升。但是，人们耗费了巨大的人力和财力，得到的却是错误百出的产品，就连世界上最精心设计

的，花费了巨额投资的阿波罗登月计划也毫不例外：阿波罗8号太空飞船，因为程序中的一个错误，使存储器中一部分信息丢失；阿波罗14号在10天的飞行中，程序竟出现了18个错误；由于FORTRAN程序中的DO语句中写错了一个逗号，造成第一次飞向金星的失败。这一期间，可以说"没有不出现错误的软件"。这一现象导致人们对软件生产的普遍悲观情绪。IBM公司的OS/360系统的研制负责人Brooks生动地描绘了当时研制过程的情景："……象巨兽在泥潭中作垂死挣扎，挣扎得越猛，泥浆就沾得越多，最后没有一个野兽能逃脱淹没在泥潭中的命运。……程序设计就象是这样一个泥潭，……，一批批程序员在泥潭中挣扎，……没有人料到问题竟会如此棘手。"

这种现象就是所谓的"软件危机"。为了摆脱这种困境，软件工作者们认真地分析了"软件危机"产生的原因，经过多年的探索与实践终于搞清楚：

(1) 程序的效率主要由算法的复杂性决定。例如，在运算速度为每秒一百万次的计算机上处理 $10\times10cm^2$ 的人造卫星摄制的底片，取点距离 $1\mu m$，用快速傅立叶变换算法处理只需要1小时，而用传统的傅立叶变换则需要三个世纪。一般说来，算法的效率从数量级上决定了程序的效率，"程序不可能比算法更好"。算法一经确定，靠程序技巧挣来的效率已微不足道，尤其是在计算机内存容量和运算速度已大大提高，计算机时费已相当便宜的今天，花大量的人力、财力去挤一点效率的"油水"，实在太不合算了。

(2) 随着软件规模日益庞大与复杂化，程序的可靠性问题越来越突出。早期以追求效率为主要目标的程序设计思想所产生的程序可读性低，可靠性差，难以维护。为了提高程序的可靠性和维护性，必须提高程序的可读性。这就要求人们在进行程序设计时，必须从"效率第一"转向到"清晰第一"。或者说，好的程序设计风格产生的程序应当是清晰易读，可靠性高，可维护性好的程序。

(3) 程序生产要从根本上摒弃以前那种随心所欲的状况和手工业方式，使之科学化，规范化，工程化。这样不仅能保证程序的质量，还能提高软件的生产效率。于是人们开始系统地、科学地研究程序设计方法以及组织软件生产的工程化方法。于是，程序设计方法学与软件工程学便应运而生。在这些方法中，与本课程关系最密切的是：结构化程序设计和程序设计风格。

结构化程序设计是从程序的结构和实施方法上讨论如何产生可读性好、可靠性高、易于维护的程序。而程序设计的风格是要求程序员有一个好的习惯。

6.2 结构化程序设计

结构化程序设计(Structured Programming)是荷兰学者E. W. Dijkstra等人在研究了人的智力的局限性随着程序规模的增大而表现出来的不适应之后，于1969年提出的一种程序设计方法。这是一种组织复杂任务时避免混乱的技术。它针对前一时期程序设计界的"自由化"倾向提出了把程序结构规范化的主张，要求对复杂问题的求解过程应按人们大脑容易理解的方式进行组织，而不是强迫人们的大脑去接受难以忍受的冲击。具体地说，它包括以下三方面的内容：

1. 程序由一些基本结构组成。由这些基本结构顺序地构成一个结构化的程序。在1.2节中已经介绍了三种基本结构，即顺序结构，选择结构和循环结构。由这三种基本结构又可

派生出多分支结构。这些基本结构如图 6.1 所示。其中(a)为顺序结构;(b)为二分支选择结构;(c) 为多分支选择结构,也称 CASE 结构,(d)为 WHILE 型循环结构;(e)为 UNTIL 型循环结构。

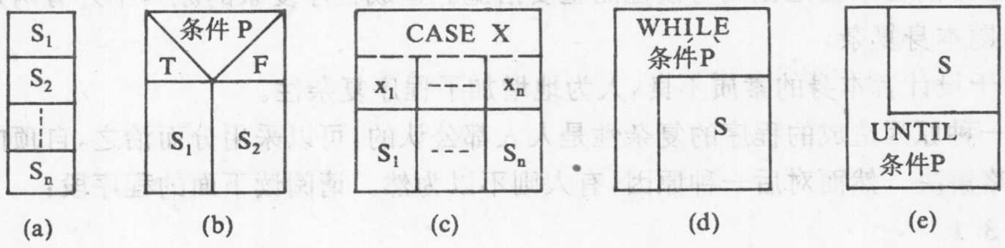


图 6.1

2. 大型程序应分割成一些功能模块,并把这些模块按层次关系进行组织。

3. 在程序设计时应采用自顶向下逐步细化的实施方法。

按结构化程序设计方法设计出的程序,结构良好,各模块间的关系简单,每一个模块内都由基本单元组成。这样的程序清晰易读,可理解性好,容易设计,容易验证其正确性,也容易维护。同时,由于采用了"自顶向下、逐步细化"的实施方法,能有效地组织人们的智力,有利于软件的工程化的开发。关于结构化程序设计的知识,在前面的章节中已分别作了介绍,这里不再赘述。

6.3 程序设计的风格

前已述及,初期的程序设计基本上可以看作是程序员的一种艺术创造,后来随着程序规模的增大,原来的一套指导思想和技巧,已经不能满足要求,人们开始从科学的角度来研究程序设计中的基本原理和方法。从 60 年代起,人们逐渐总结出一些规律。到了 70 年代,它已经上升为计算机科学的一门学科。然而,就目前的情况看,为计算机编程序仍然是一项带有很大手工劳动色彩的智力密集劳动。也就是说,它既是一门科学,又保持着艺术的特点。常常遇到这样的情形:把同一个问题的程序交给受过同样程序设计训练的人分头独立地去完成时,设计出来的程序之间的差别,竟会达到令人吃惊的地步。这种差异与每个人的智力情况和习惯有很大的关系,通常把一个程序员在程序设计过程中,在程序的各个方面,从标识符的定名、程序行的排列格式直至模块分解,控制结构所表现出贯穿始终的习惯的总和称为该程序员的程序设计风格(stgle)。

实践证明,好的程序设计风格能发挥程序设计语言的长处,并弥补其不足;编写出的程序看起来美观雅致,读起来流利舒畅。这样的风格产生的程序必然是正确的、易读的、易维护的。

那么,什么是好的程序设计风格呢?对这个问题很难给出确定的回答。因为程序设计的环境,用户的要求和各人的性格习惯、兴趣等因素都会影响程序的风格。但是,这并不是说,程序设计风格就没有统一的原则可供遵循。恰恰相反,程序设计的风格是针对程序的可读性而提出来的,现在,大家都认为应当将可读性作为衡量程序设计风格的一个基本准则。围绕可读性,人们还总结出来一些行之有效的具体原则和方法。

首先应强调的一点是,采用结构化程序设计方法是获得好的程序设计风格的重要前提。

下面列举的原则和方法都是基于这一点而言的。

6.3.1 基本风格：简短朴实

降低程序的复杂性是提高可读性的重要措施。造成程序复杂的原因不外有两点：

① 问题本身复杂。

② 由于设计者本身的素质不良，人为地增加了程序复杂性。

对前一种原因造成的程序的复杂性是人人都公认的，可以采用分而治之，自顶向下逐步细化的策略解决。然而对后一种原因，有人则不以为然。请阅读下面的程序段：

例 6.3.1

```
for I=1 to N
  for J=1 to N
    let A(I,J)=Int(I/J) * Int(J/I)
  next J
next I
```

可以断定，读者要读懂这段程序是相当费力的。要是把它改写成下面的程序段就要易读多了。

```
for I=1 to N
  for J=1 to N
    let A(I,J)=0
  next J
  let A(I,I)=1
next I
```

这两个程序的作用都是建立一个单位矩阵，即把一个方阵对角线上的元素(A(I,I))全部置"1"，其余元素全部置"0"。后一个程序的意义是很显然的。而前一个程序是设计者费了一番心思才考虑出来的。其设计思想是，只有 I=J 时，Int(I/J)与 Int(J/I)才都为"1"；而 I≠J 时，Int(I/J)与 Int(J/I)中必有一个是"0"，即得非 I=J 时的 A(I,J)为"1"，其余元素均为"0"。这样的构思，可谓"巧妙"，但却人为地增加了程序的复杂性。费了九牛二虎气力，虽然可省去了一条语句，但却使得整个程序可读性变差，可说是得不偿失，并且常常事与愿违。如例 6.3.1 中的后一个程序段中，虽增加了一条赋值语句，即总共增加了 N 次赋值操作，但这 N 次赋值操作以及内循环中的 N^2 次赋值操作中都不包括算术运算。然而前一个程序段中的 N^2 次赋值操作中，每一次都需进行两次除法运算，两次取整运算，一次乘法运算。实际的运行时间反比后一个程序段长得多。这样的例子还有许多：

例 6.3.2 阅读下面的程序段：

```
let A=A-B
let B=A+B
let A=B-A
```

这一段程序只不过相当于下面的一段程序：

```
let C=A
```

```
let A=B
let B=C
```

即将 A,B 两个变量的值对调。前者虽然少用了一个临时变量,却变得迂回曲折,晦涩难懂,又增加了运算次数,可说是画蛇添足。

当然,并不否认许多程序员在技巧上所花费的代价可以转换为程序效率的提高。

例 6.3.3 某工厂每人每月平均生产某种产品 75 件,为鼓励工人多生产,工厂决定每人每天平均生产超过三件的,月终奖励 15 元;不足三件的罚 15 元;只生产三件的不奖不罚。设某人的基本工资为 Bwages,某月生产产品总数为 pn,则计算其月收入 income 的程序如下:

```
print "Base wages, prodscts";
input Bwages, pn
if pn/30>3 then
  let Income=Bwages + 15
else if pn/30<3 then
  let Income=Bwages-15
else
  let Income=Bwages
end if
print "Income is:" Income
end
```

现改成下面的程序:

```
print "Base wages, products";
input Bwages, pn
let Income=Bwages+Sgn(pn/30-3) * 15
print "Income is"; Income
end
```

这后一程序巧妙地利用了符号函数的特点,以一个语句行代替了 7 个语句行。程序的效率是可能提高的。但是,它的可读性却不如前一个程序好。在 6.1.2 节中已经讲过,在计算机硬件已相当发展的今天,程序的效率已不再是程序设计时优先考虑的因素。在多数情况下,我们宁愿牺牲一些效率,也要保证程序的可读性。对初学者来说,为了培养自己的好的程序设计风格,更不应追求过于巧妙的程序。

有些程序员不顾程序设计的风格,而热衷于卖弄小"技巧"。以为程序愈晦涩难懂,自己水平愈高。孰不知这样给测试、纠错和维护带来极大的困难。这种程序设计风格是不可取的。

总之,程序中应该尽量采用简单的典型的算法,不要使用华而不实的古怪技巧。与此同时,还应注意程序要尽量短,不要兜圈子。因为冗长是错误"庇护所"。

6.3.2 程序文档化

要使一个程序具有较好的可读性,除要求程序结构清楚,借助于文档等条件外,另一个可靠的措施是使程序文件(documentation)化。也就是说,一个程序要象一份文件那样,格式

清晰、容易阅读。这里讲的程序文档化主要包括以下三方面的内容：

(1) 充分、准确、合理的注释；

(2) “见名知义”的标识符；

(3) 清晰的书写格式。

分述如下：

1. 注释——阅读程序的向导：注释是程序员对程序作的语化与解释，以向读者提示程序的功能、设计思想、用法等信息，也可以说是程序员给阅读者提供的阅读向导。在2.1.2节中曾指出，注释可分为序言性与功能性两种，序言性注释通常置于每个程序单位的开头，向读者指出本程序单位的功能和目标；本程序单位与其它程序块的接口；所引用的标识名的意义、用途、约束；开发的日期、人员及有关历史等。功能性注释通常嵌在源程序中，对程序的各部分加以说明、解释。

从提高源程序的可读性的角度看，注释是一个程序的重要组成内容。几乎所有的程序设计语言中都引入了注释这一概念。

有人认为，程序的源代码是硬指标，注释是软指标，是可有可无的。这是对程序的可读性还没有充分认识的表现。如在例6.3.3第二个程序中，如果在第三行前，加以适当的注释，便可增加其可读性，这样的程序并不是不可以使用的。

还有人觉得，在编写程序时，还要考虑写注释太费事。但是，如果不加注释，会使别人理解程序和维护程序增加许多困难。事实上，在写程序的过程中，顺手加进几句注释，是并不困难的，用英语或汉语拼音均可以。更可喜的是，现在许多计算机程序设计语言都已汉化了，完全可以给自己的程序加上用汉字写的简短注释。

那么，如何使用注释呢？简单地说就是：充分、准确、合理。

在程序中总是给费解的地方加上注释。但在实际编程序时，由于程序员已对程序结构和各个细节以至各条语句经过一番深思熟虑，很容易觉得每一部分都比较清楚，不需要注释，因此给出的注释往往偏少。针对这一倾向我们提出了注释要充足：只要需要就应给出注释，而且还要“更多”些。True BASIC 程序中附加的注释不会影响其执行速度。因此，完全可以放心地去写注释，把程序写成一个易读的程序。

当然这并不是说，可以随意给程序加注释。好的注释可以为读者提供从代码本身得不到的一些信息。然而，错误的注释却会把人引入歧途。所以与其给出错误的注释，倒不如不给注释。

例 6.3.4 一个错误的注释：

```
  input x
★if x<0 then! Text for negative value of y
  print "The value of x must be";
  print "greater than zero."
  else
  ……
```

这一段程序中，带“★”标记的程序行中的注释是错误的。因为程序中并不存在一个变量“y”，而且还把“Test”写成了“Text”。幸好这个例子中的注释不会引起别的误会。再看另一个例子。

例 6.3.5

```
! This time we shall test for
! odd numbers
if Mod (x,2)=0 then
    let Sum=Sum+x
    let Oddno=Oddno+1
end if
```

在这段程序中，注释告诉我们校验奇数，而且变量名也促使我们相信这一点。但条件语句中却是选取了偶数。这很容易引起读者的混乱；到底哪个对？因此一定要确保注释与程序一致。

另外，要注意让注释能给出程序代码本身给不出的信息，不要把注释变成程序的重复叙述。例如：

例 6.3.6

```
:
print x      !   print value of x
:
```

这样的注释是毫无意义的。

2. "见名知义"的标识符：注释虽能够有效地提高程序的可读性，但它毕竟是从外部附加到程序上的。从程序内部来说，一个程序要用到大量的标识符——如变量名、数组名、函数名、子程序名、文件名等。如果程序中所有用到的标识符都选择得比较合适，就可以提高源程序本身的可读性，而不必完全(或主要)依赖于注释。

给标识符命名应遵守以下原则：

(1) 标识符要按意取名，即要能让读者"见名知义"。如果一个程序中使用了大量的代数符号如 A,B,C……,A1,A2,A3,……，变量一多，人们便不易弄清它们到底各代表了什么意义。按意取名可以较好地解决这一问题。如学生年龄用 Student-age，学生性别用 Student-sex，……等，人们看到这些变量名，不需作任何解释便可知道它们的含义。对函数名、子程序名来说，最好按其功能命名。

True BASIC 允许数据名、函数名、子程序名的有效长度多至 31 个字符。这对命名十分有利。容易做到名字与自身的含义相一致。

(2) 要选择含义准确的英文字或汉语来命名，不要写错，如例 6.3.3 中把"Bwages"错写成"Bwagers"将引起更大的混乱。

(3) 尽量避免不明确的缩写。必要时在程序中以注释形式给出一份缩写与全称的对照表。

(4) 对逻辑上有关的数据名或同一模块中的函数名与子程序名应采用相关的一类名称。

3. 清晰的书写格式：有人说，程序设计就象赋诗。好的程序就象一首优美的诗歌。要构造一个好的程序除了要求有好的算法和好的程序设计方法外，清晰的书写格式也能为其增添不少光彩。

好的书写格式是通过缩格和空行实现的。许多语言都允许程序中留有空行，以便明显地将程序中的不同部分分隔开，如重要的注释前后留有空行可以使注释更加突出；在子程序和函数前后留有空行，使子程序或函数的范围更为分明。

程序编码中正确地运用缩格，可以突出程序的逻辑层次结构。

在例 2.5.1 中曾经给出了如下书写形式的一个从三个数中找一个最大数的程序：

```
!   Find the maximum from 3 numbers.
!
input prompt "a, b, c=":a, b, c
if a>b then
  if a>c then
    let max=a
  else
    let max=c
  end if
else if b>c then
  let max=b
else
  let max=c
end if
print "max="; max
end
```

这里，条件嵌套的层次关系十分清晰。如果改成下面的书写格式，便会黯然失色，给读者理解程序增加不少困难。

```
! Find the maximum from 3 numbers
input prompt "a, b, c"; a, b, c,
if a>b then
if a>c then
let max=a
else
let max=c
end if
else if b>c then
let max=b
else
let max=c
end if
print "max"; max
end
```

如果问读者"这两个程序你喜欢哪一个呢？"我们相信你是会选择前者的。

这里还要说明的是，我们指的缩格书写格式的前题是要正确地反映程序的逻辑层次结

构。下面是上述程序的另一种缩格排列形式：

```
if a>b then
   if a>c then
      let max=a
else
   let max=c
   end if
else if b>c then
   let max=b
else
   let max=c
   end if
   print "max="; max
end
```

它虽然采用了缩格书写格式，却比不采用缩格书写方式更坏。因为它没有正确地反映程序的层次结构，因而有可能使读者更难以理解程序的逻辑。

有人会说，缩格书写方式固然好，但效率低，要多占用存储空间。这句话对早期的 BASIC 程序来说，确是这样，每增加一个空格要多占一个字节的存储空间，并影响执行速度。但是，对 True BASIC 却不存在这个问题。如前面已经介绍过的，True BASIC 程序中附加空格不会影响执行速度。这一点使我们可以放心地去书写易读性好的程序。即使对旧的 BASIC 程序来说，增加空格虽然会使效率受到一些影响，但按"清晰第一，效率第二"的原则，这样做是值得的，是利大于弊的。

4. 书写文档：在计算机出现的早期，程序的规模比较小，而且大多是专用的。人们拿来就用，一般没有什么问题。那时，人们就把一个计算机上的程序系统称为其软件。

后来，计算机的应用范围和复杂程度急剧膨胀，而且程序的通用性增强。用户只拿到程序，没有说明书是难以入手的。同时，维护的地位也日益上升。而要对程序进行维护没有资料更是十分困难的事情。所以这时，程序及其有关文档资料便是不可分割的了。于是"软件"的概念也应予以更新。通常认为软件应有三部分组成：一是软件的可执行部分——程序；二是程序运行时所需的数据；它是与程序相辅相成的，在运行中不可缺少的。三是与程序有关的文档资料，它不是运行程序所必须的，但却是理解程序所必须的。

从软件工程的角度看，一个软件从作为项目提出并着手实现，到该软件报废或停止使用（也称软件生命周期）要经过规划、需求分析和定义、软件设计、编码、验证、测试和维护等几个阶段。每一个阶段都应产生相应的文档，程序设计仅仅是软件生命周期中的一个局部。主要涉及软件详细设计、编码及验证等几个阶段。因此，在程序设计过程中，也要注意整理有关资料，写成有关文档，如算法说明、流程图、数据结构的定义、程序说明书等。文档资料是理解程序和维护程序所必须的。

过去那种只注重程序，而不注重写文档资料的作法应当纠正。

6.3.3 使用过程的具体原则

函数和子程序通称为过程。对大型程序来说，过程常常是分解程序的基本手段。人们阅

读程序也是以过程为单位进行的。所以过程的划分，对程序的可读性有很大的影响。

1. 每个过程只反映一个概念：在第四章中已经讲过，模块分解可以有两种准则：一种是按功能划分，要求每个模块只完成一种功能；另一种是按数据划分（如对数据库的操作），要求每个模块只对某一些数据操作。这两种分解准则的共同点都是每个模块只反映一个概念。在设计的程序中，多数按功能划分比较合理。按功能划分模块，就是说每个过程只做一件事，而且一件事只放在一个过程中去做。在一个过程中安排几件事，会增加模块的复杂性，降低可读性。反之，把一件事分在几个过程中去做，无疑会使程序冗长，重复。

2. 过程应是自封闭的：所谓过程是自封闭的（Self-contained），即过程中的变量应是局部的，仅通过参数与外部环境打交道。为此要限制全局变量的数目。

3. 限制每个过程的参数的数目：使用外部函数和子程序时，过程仅通过参数与外部打交道。如果一个过程有太多的形式参数，就表明过程与调用模块之间联系的因素比较多，也就是说调用模块时需要考虑较多的条件，过程执行结束以后对调用模块也可能产生较多的影响。这样会增加设计、调试、理解和修改程序的困难。

根据某些心理学家的研究，一个人一般能同时处理 5～9 件事务，超过这个限度，就容易顾此失彼，程序设计是一项比较复杂的工作，一般认为过程参数以 3～5 为宜。但这也不是绝对的。

4. 一个过程内的可执行语句一般不要超过 24 行：通常人们习惯于一个过程内的语句不超过 24 行，因为这个数目内的程序易于阅读、理解和记忆，并且可在屏幕上的一帧上显示或打印在一页内。不要怕过程太短。只要功能独立，就应把它划分为一个过程。

6.3.4 使用控制结构的具体原则

控制结构是决定程序中语句或语句组之间执行顺序关系的语法结构。结构化程序设计提出了以顺序、选取、循环三种规范化的控制结构组织程序的基本要求，不允许程序员任意地使用转移，使每一条语句都只有一条路径可以达到。因此，整个程序可以从头读到尾，不需前后跳转。

但是，这仅仅是一些基本原则。在实际应用中，有时不可避免地要突破这些原则。例如，在 True BASIC 程序中的选择、循环结构中，允许使用 EXIT LF、EXIT DO 或 EXIT FOR 语句。EXIT 语句的使用，将会使这些基本程序结构呈现多出口状态。但是，有时不用，将会使程序的效率明显地降低或可读性降低。在这种情况下，就需要有一些更具体的原则，使程序设计能根据所用语言的特点，扬长避短，形成好的风格。下面针对 True BASIC 的特点提出一些较具体的原则：

1. 避免没有 ELSE 部分的选取结构：LF——THEN——ELSE——END IF 结构能给予读者一种心理上的平衡。选取结构中当然可以没有 ELSE 部分，但这将会使程序失去平衡，容易使人产生误解。

在例 2.6.9 中，曾给出了如下五家共井的程序：

```
for x=50 to 1000
  let c=148 * x/721
  if c=Int(c) then
    let d=x-4 * c
```

```
        let e=x-5*d
        let a=x-6*e
        let b=x-2*a
        print "x="; x
        print "a="; a, "b="; b, "c="; c, "d="; d, "e="; e
      end if
    next x
    end
```

读者不容易找出 C≠Int(C)时应执行什么操作。

2. 避免使用 THEN IF:在 THEN 块(THEN 后的语句)中出现 IF——THEN 结构时，会使得在处理 THEN 块后面的 ELSE 语句究竟与哪一个 IF 语句配对问题上出现混乱。

前面曾列举过例 2.5.1 中的程序的另一种写法：

```
  input prompt "a,b,c="; a,b,c
★if a>b then
★if a>c then
  let max=a
  else
  let max=c
  end if
  else if  b>c then
  let max=b
  else
  let max=c
  end if
  print "max="; max
  end
```

这一程序难读的原因除了书写上的毛病外，还在于它使用了 THEN IF 结构(有"★"标志的语句行)。读者要弄清后面的 ELSE 语句属哪一层次是要费一番心思的。

有人说，这不要紧，只要按缩格格式书写不就可以了吗?是的，但"智者千虑，必有一失"。倘若书写时，写成下面的缩格形式，那就更糟了：

```
input prompt "a,b,c="; a,b,c
if a>b then
  if a>c then
  let max=x
else
  let max=a
else if
  let max=c
end if
else if b>c then
    let max=b
```

```
else
    let max=c
end if
print "max="; max
end
```

结果更把读者引入了困境。因此最好的办法是避免使用 THEN IF 结构。

3. 不要嵌套太深：控制结构的嵌套是结构化程序设计中积木式程序设计思想的一种体现。然而，每进入一个内层嵌套，都要求程序员或程序的读者记下上层处理过程中的一个头绪。于是随着嵌套层次的增加，要求人们记下的头绪越多。如前所述。由于人并行处理事务的能力有一个生理上的极限，因此嵌套过多，会增加程序的复杂性。一般来说，嵌套深度以三至四层为宜。

4. 有限制的使用 GOTO 语句和 EXIT 语句：GOTO 语句曾在程序设计史上引起过一场轩然大波。结构化程序设计就是在这场争论中诞生的。使用 GOTO 语句要求程序有标号。True BASIC 规定，只要每条语句都加以行号(按书写顺序由小到大)，便可以使用 GOTO 语句。

例 6.3.7 阅读下面的程序：

```
20 input m,n
30 let r=mod(m,n)
40 if r=0 then
50 print n
60 goto 99
70 let m=n
80 let n=r
90 goto 30
99 end
```

读者读完上面的程序有什么感觉呢？你会觉得很不习惯。甚至觉得它有些混乱。

可是，早期的程序员对 GOTO 语句却非常熟悉，而且使用 GOTO 语句是他们的拿手好戏。GOTO 语句用法很灵活，它可以借助于标号从程序的一处将流程转向另一处，程序员常借它创作自己的"艺术品"，编写出高效率的程序。然而 GOTO 语句却是破坏程序清晰的一个祸根。

1963 年 1 月 Peter Naur 指出：如果你仔细地观察，将会惊奇地发现：向后跳转的 GOTO 语句通常是一个循环的隐蔽形式。而且你将愉快地看到，当你用 FOR——NEXT 语句代替这种 GOTO 语句时，算法的清晰性就得到了明显地改善。

1965 年荷兰学者 E. Dijkstra 提供了如下情况："来自不同国家并有不同背景的两个程序设计部门(一个主要是科学计算，另一个主要是商业)的管理人员给我写信，它们彼此独立地根据各自的创造性发现，程序的质量同程序员在程序中使用 GOTO 语句的密度成反比，……。对所试验的程序，没有 GOTO 语句的程序显得更短且更易懂"。读者将例 6.3.7 中的程序与例 2.6.4 中的程序对比一下，立即会明白这一点。

1968 年 Pijkstra 提出了"GOTO 语句太原始，是造成程序混乱不堪的祸根，应当把它从

所有的高级语言中消灭掉"的论断。这一论断立即在软件界引起一场论战。逐场讨论,加速结构化程序设计方法的充实、发展、完善和广泛应用。并且一致认为,GOTO 语句还是有用的,但也是有害的。在程序设计中不能一概禁止,但要严格限制它的使用。应该少用,最好不用。

True BASIC 只有使用行号时,才允许使用 GOTO 语句。这是为了让那些习惯了旧 BASIC 的人能使用 True BASIC。True BASIC 的本意是废弃 GOTO 语句。在非用 GOTO 语句不可的场合,它提供了一种规范化的 GOTO 语句——EXIT 语句。

EXIT 语句比 GOTO 语句规范化之处在于它只能从一个控制结构内部转出这个控制结构,即唯一地转到该结构的出口处。

5. 不在 FOR 循环体中给循环变量赋值:FOR 循环是一种计数型循环,循环的次数由循环变量的初值、终值和步长决定。当循环体内存在给循环变量赋值的操作时,便会破坏预定的循环次数,降低程序的可读性。

6.3.5 提高表达式的可读性

表达式是程序的重要组成部分。表达式的可读性是影响程序可读性的重要因素。程序可读性对表达式的基本要求是清晰、明了。书写表达式的好的风格是:

1. 使用括号,避免误解:通常表达式中的运算顺序是由各运算符的优先级控制的,也可以由括号强调运算的优先级。按理,不用括号能表达的,就不一定要用括号,而且有些程序员常常会"巧妙"地省去一些括号。但是,仅靠运算规则常常会引起一些误解。如

    A * B/2 * C

有时人们会将它理解为

    (A * B)/(2 * C)

为了避免发生这种混乱,就应使用括号,强调一下运算的顺序。如改为

    ((A * B)/2) * C

所以,在书写表达式时,只要有容易产生误解的地方,就应使用括号。再如

    A OR B+C

很容易误解为

    (A OR B)+C

但实际上是

    A OR (B+C)

使用了括号减少了误解。

2. 使用自定义函数代替重复表达式:函数的使用可以提高程序的清晰度。而且如果有标准函数可用,应尽量使用标准函数。无标准函数可用时,可自定义一些函数。

例 6.3.8 已知平面坐标系上的三点 A(x_1,y_1),B(x_2,y_2),C(x_3,y_3),求三角形 ABC 的三条边长。

求解该问题的程序可以包括下面的程序段:

```
⋮
! Compute lenghs of sides.
!
```

```
let AB=Sgr((xz-x1)^2+(y2-y1)^2)
let AC=Sgr((x3-x1)^2+(y3-y1)^2)
let BC=Sgr((x3-x2)^2+(y3-y2)^2)
⋮
```

如果定义一个函数，程序的可读性便会提高不少：

```
⋮
def side(XA,YA,XB,YB)=Sgr((XB-XA)^2+(YB-YA)^2)
let AB=Side(x1,y1,x2,y2)
let AC=Side(x1,y1,x3,y3)
let BC=Side(x2,y2x3,y3)
⋮
```

这样不仅可以提高编程序效率，也能提高程序的可读性。

3. 通过逻辑变换改写条件表达式：由 AND,OR,NOT 组成的条件表达式有难以理解之处，应变换为容易理解的形式。如

将 if not A and not B then……改写为 if not (A or B) then……；

将 if not A or not B then……改写为 if not (A and B) then……；

将 if not XX then A else B 改写为 if XX then B else A；

将 if XX <> YY then A else B 改写为 if XX=YY then B else A 等。

这样，比较符合人的思维习惯，从而提高了表达式的可理解性。

习　　题

6.1.1　学习完 6.1 节后，小王与小李展开了一场争论。小王说："正确的程序，一定是可靠的。"小李说："可靠的程序，一定是完全正确的"。你说他们俩的话谁说得有道理？为什么？

6.1.2　为什么还要对编好的程序进行维护？一般有哪些种类的维护？

6.1.3　有人说程序编好后只要能上机运行就行了，为什么还要讲可读性呢？你觉得对吗？为什么？

6.1.4　程序的效率重要呢？还是清晰重要？为什么？

6.1.5　你认为什么样的程序才算好程序呢？

6.2.1　程序的三种基本控制结构有何共同特点？使用这三种基本结构编写程序有何好处？

6.2.2　把大的程序划分为模块有何好处？是不是划分的模块越少越好？

6.2.3　什么叫"自顶向下、逐步细化"？

6.3.1　程序设计的方法与风格有什么区别与联系？

6.3.2　程序长一点好呢？还是短一点好？

6.3.3　程序中的注释是否越多越好？

6.3.4　一份编得不好的程序加上注释后，是否可以变得好一点？

6.3.5　有人认为边写程序边加注释容易分散程序员的注意力，他主张把程序写完后再集中精力去写注释。这种方法可取吗？

6.3.6 好程序需要的注释多，还是差程序需要的注释多？

6.3.7 标识符取得长一些好，还是短一些好？

6.3.8 是不是很小的程序就可以不用缩格书写格式？

6.3.9 什么是软件？

6.3.10 文档有何用处？

6.3.11 若几个过程都很小，可不可以把它们合并成一个过程？

6.3.12 嵌套太深有何坏处？

6.3.13 “结构化程序设计禁止使用 GOTO 语句”这句话对吗？

6.3.14 如何提高表达式的可读性？

6.3.15 试举出五个风格不好的程序的例子。

第七章　数据的输入与输出

程序是为使用者设计的。对用户来说，程序只不过是能执行特定功能的“黑箱”(black box)，用户只须知道它需要输入什么以及能获得输出什么即可，而不一定很关心其内部结构。这里所说的“输入”与“输出”，不仅包括了输入数据和输出数据，还应当包括输入/输出风格。输入/输出风格，决定了用户对程序可以接受的程度。从“用户第一”的观点来看，输入与输出设计是影响程序质量的重要因素。通常评价程序的输入/输出风格的主要标准是可用性(availability)和健壮性(robustness)。如前所述，可用性要求程序体现“与用户友好”的原则；健壮性要求对于任意的输入都应有相应的输出。

为获得良好的输入/输出风格，程序应当具有灵活多样的输入/输出方式与格式。本章和第八章将在前面介绍过的程序的基本输入/输出语句的基础上，补充一些其它的输入/输出语句。此外，还要介绍打印机、显示器的基本使用知识及 True BASIC 的初级作图功能。

7.1　数据的输入

7.1.1　输入风格

在进行程序设计时，为了获得好的输入风格，一般应考虑如下几条原则：

1. 要检验输入数据

1972 年 6 月，美国《计算机世界》报曾报道了一则消息，因一位计算机操作员错敲了一个键，使某市的税收损失了二十九万美元。这一教训告诉人们，一个好的程序应当检查输入数据的合法性与合理性。

合法性检查一般是指数据类型的检查。例如对语句

```
input name$, age, sex$
```

如果输入

```
123, Zhang, 52
```

便是输入了不合法的数据。因为姓名一般不会叫“1 2 3”；年龄应是数字，不应是字符；性别不应是数字。

合理性检查(reasonableness check)是指对数据是否在所规定的范围之内的检查。例如输入一个人的工龄为“25”，年龄为“23”，那么这两个数据便是不合理的。因为工龄绝不可能大于年龄。再如，曾举过一个例子：给定三个数(A, B, C)，希望求出用这三个数作为边的三角形的面积。三角形的三条边应满足以下条件：任何两边之和应大于第三条边。因此，对输入的三个数必须检查是否符合“两数之和大于第三数”。如果不满足此条件，则无法计算面积。然而，仅符合“两数之和大于第三数”还是不够的。如输入的三个数为－3，－4，－5，虽然满足上述条件，但是也无法组成一个三角形。

计算机界有一句谚语:“输入是垃圾,输出也是垃圾”。除了要对数据的合法性,合理性进行检查外,还应检查数据的正确性。如输入某人的高考成绩时,将520分,输成250分,将使一名优秀学生失掉一次升学机会。

2. 要利用数据终止标志或函数来结束输入,避免通过计数来结束输入

FOR循环是一种计数型循环。下面是一种常见的数据输入形式:

```
for i=1 to n
  read a(i)
next i
```

这里n是由人给出的。人要通过数据的个数才能知道n的确切值。如果数据很多,一个一个地去数数据不仅很麻烦,而且可能出错。最好是把数数据的工作交给计算机去做。因为机器比人更善于做这种重复繁琐的工作,而且会有效而正确地去完成它。程序设计者可以在程序中设定一个标志作结束标志,在执行程序过程中,每输入一个数据就对这个标志进行一次检查,判断它是否结束标志,如果是,则表示数已读完,否则,应继续输入数据并进行相应的处理。如上述程序可改写为:

```
let i=0
read x$
do while x$<>"done"
  let i=i+1
  let a$(i)=x$
  read x$
loop
```

每次输入一个字符串给字符串数组中一个元素,直到输入字符串“done”为止,另外也可以通过数据函数来测试是否还有数据可读。True BASIC 提供的数据函数有:MORE DATA 与 END DATA。

例 7.1.1 一个使用函数 MORE DATA 的例子。

```
! Sum up all number in the DATA statements.
! Print the result.
!
let Sum=0                 ! initial sum is zero
do while more data        ! loop while there's more
  read X
  let Sum=Sum+X
loop
print "Sum is"; Sum       ! print the sum
data 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
end
```

```
Ok. run
Sum is 55
```

Ok.

MORE DATA 函数能自动检查数据是否读完,MORE DATA 的值为"真"表示还有未读的数据。

3. 在输入数据时应给用户以必要的"提示"

用 INPUT 语句由键盘输入数据时,如果不给用户以必要的提示,而仅仅显示一个"?",常常使用户茫然,不知此时应输入哪个数据。譬如一次要求输入的数据量很多时,操作人员可能记不住应输入多少个数据以及按什么顺序输入数据,往往需要查程序中 INPUT 语句才能确定,这是操作人员最感头疼的事情。若能用 INPUT PROMPT 语句或 PRINT 语句给出必要的提示,将会减少输入时的错误,并使用户有一种"实在感"。

4. 将输入置于一个独立的模块中

输入是程序中比较明确的一个功能。这一功能应当由一个模块来实现它。同时,将输入部分置于一个模块时,当需要修改输入格式时,会非常方便。尤其是用 READ 语句进行输入时,把 READ 与 DATA 放在一个模块中,当需要修改数据时,仅涉及输入模块本身而不涉及其它模块,增强了程序的通用性。

以上介绍了获得好的程序的输入风格的几条主要原则。下面补充几条能有助于获得好的输入风格的输入语句和函数。

7.1.2 单键输入(GET KEY)语句

GET KEY 语句能从键盘缓冲区中取一个字符,得到该字符的 ASCII 代码。格式为

GET KEY〈变量〉

使用 GET KEY 语句应注意以下几点:

(1) 如果用户按下的键有相应的 ASCII 码,则 GET KEY 得到该键的 ASCII 码(参见附录 D.1);否则 GET KEY 得到的是 ASCII 码范围之外的机器码(参见附录 D.2)。

例 7.1.2 是否选择:程序的输入方法应考虑使得用户花费的代价最少。"是否选择"便是这样的一种方式。当程序含有两种功能,需要用户从中选择一种时,不是让用户去输入功能各本身,而是通过程序提问,让用户简单地回答"Y"("y")或"N"("n")。

下面是一个"是否选择"的程序段:

```
⋮
print "继续运行吗? (Y/N)"
do
  get key key
  if key=089 or key=121 then
    enit do
  else
    if key=078 or key=110 then stop
  end if
loop
⋮
```

这里 089 是字符“Y”的 ASCII 码，121 是字符“y”的 ASCII 码。只有按下的键是大写“Y”或小写“y”时，才能脱离循环，转向执行，其下面的程序段，如果按“N”或“n”键，程序结束运行。

(2) 输入的字符不显示在屏幕上。

例 7.1.3 输入口令：口令是防止程序被非法使用的一种手段。为了能具有较好的保密功能，合法用户在输入口令的过程中，不希望别人看到他输入的内容，即希望不在屏幕上显示出口令来，下面是一个口令检查程序段。

```
⋮
print "请输入口令："
let i=0
let f1, f2, f3, f4=0
do while i<3
  get key k1
if k1=071 then let f1=1
  get key k2
  if k2=111 then let f2=1
  get key k3
  if k3=111 then let f3=1
  get key k4
  if k4=100 then let f4=1
  if f1 * f2 * f3 * f4=1 then exit do
    else let i=i+1
  print "口令错误，请重新输入："
loop
if i>=3 then
  print "你是非法用户！"
  end
else
  print "现在开始使用程序……"
end if
⋮
```

程序中合法的口令是“Good”，只有用户连续打对了这四个字母，才准其使用程序。由于合法用户也会因误操作打错口令，程序提供了三次输入口令的机会。

(3) GET KEY 取的是键盘缓冲区中的字符代码。程序碰到 GET KEY 时，若键盘缓冲区中无输入的字符代码(即用户没有按任何一个键)，机器将等待，直到用户按下一个键为止。

7.1.3 测试按任一键(KEY INPUT 函数)

程序执行到 INPUT 或 GET KEY 时便暂停，等待由键盘输入数据或一个键的代码。然而有时为用户更方便，可以允许用户以按任何一个键作为程序是否继续执行的条件。

KEY INPUT 函数可以检查用户是否已经按下一个键。只要用户按下任何一个键，KEY INPUT 就为真。

例 7.1.4

```
⋮
print "按任意键则继续"
do
loop until key input
⋮
```

程序不断在执行循环(只是为了等待)直到按下一个键，才继续执行下去。这往往用于需要检查上一段的运行结果有无问题，如经检查无问题，则令其继续执行。

7.2 数据输出格式的控制

数据的输出格式是面对用户的数据的表示形式。输出格式是否得当，影响数据表示的清晰性。通常，为不同的应用目的，需要不同的输出格式。

在 2.4 节中，介绍了 PRINT 语句的基本输出格式。它能在一行内按分区格式或紧凑格式显示数据，也可以打印空行，使显示的数据之间隔开一定的距离。这一节将介绍如何改变显示区域的宽度和屏幕的最大边界；如何在一行内的任意位置上输出数据，以及如何按照需要的特定格式输出数据。

7.2.1 显示的区宽和边界

如前所述，在 PRINT 语句中，如果各数据表达式之间以分号分隔，则输出项之间按紧凑格式显示，即后一项紧靠前一项显示。若数据表达式之间以逗号分隔，则输出项之间按分区格式显示。

一般情况下，机器自动设置一行输出的(margin)为 80 个字符，每一区域(zonewidth)是 16 个字符，如图 7.1 所示。

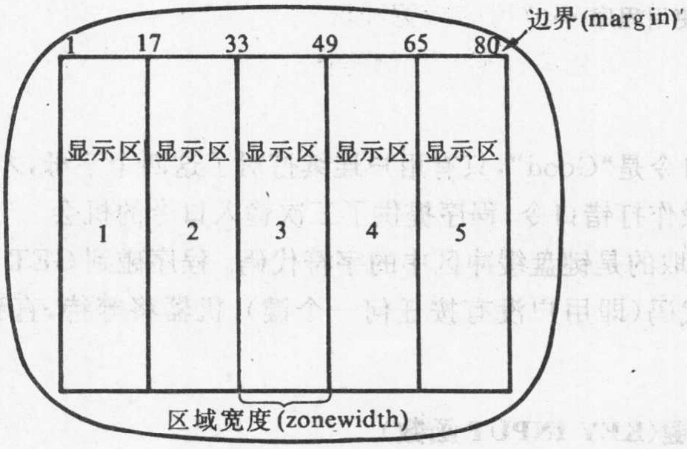


图 7.1

输出行的宽度不能超过边界。在一行上输出的内容超过边界时，剩余的输出的内容将在下一行输出。但是，True BASIC 允许用 SET ZONEWIDTH 改变输出宽度，也允许用 SET MARGIN 语句改变边界位置，即改变行宽。它们的格式为

SET ZONE WIDTH〈数值表达式〉

SET MARGIN〈数值表达式〉

应当注意：

(1) 行宽与区宽不能少于一个字符，其最大值没有限制；

(2) 行宽不能小于区宽；

(3) 设置行宽时要考虑到实际使用的外部设备实现的可能性。

例 7.2.1

```
★set margin 0
end
```

```
Ok. run
MARGIN less than zonewidth. (出错信息)
```

由于系统自动设置的区宽为 16，所以当设置屏宽为"0"时，系统首先检测出"行宽(边界)小于区宽"的错误，光标停在"★"处。下面的一个程序先把区宽设置为"0"，再把屏宽设置为"0"，这时虽不会发生"行宽小于区宽"的错误了，但首先检测出"区宽超出范围"的错误。光标停在"★"处。

```
★set zonewidth 0
  set margin 0
  end
```

```
Ok. run
ZONEWIDTH out of range. (出错信息)
```

有时操作人员弄不清当前所用的行宽和区宽，可以用 ASK ZONEWIDTH 语句查询区宽，用 ASK MARGIN 语句查询行宽。它们的格式为

ASK ZONEWIDTH〈数值变量〉

ASK MARGIN〈数值变量〉

当 True BASIC 执行完 ASK 语句后，将把区宽或行宽的字符数送到相应的数值变量中。

例 7.2.2

```
for i=30 to 120 step 30
  set margin i
  ask margin Marg
  print "margin="; Marg
```

```
next i
print
for i=1 to 30 step 5
  set zonewidth i
  ask zonewidth zone
  print "zonewidth="; Zone
  print 1,2,3
next j
end
```

```
Ok. run
margin=␣ 30
margin=␣ 60
margin=␣ 90
margin=␣ 120

zonewidth=␣ 1
␣1  2    3
zonewidth=␣ 6
␣1  2          3
zonewidth=␣ 11
␣1          2           3
zonewidth=␣ 16
␣1                   2         3
zonewidth=␣ 21
␣1                        2           3
zonewidth=␣ 26
␣1                             2           3
Ok.
```

在输出数据时，若指定的区宽为 1 时，由于一个数值至少占三列，超过了所指定的区宽为 1 的范围，系统按实际需要增加了区宽。

7.2.2 TAB 定位

在 PRINT 语句中使用 TAB(x)能使其后面的输出项在第 x 列的位置开始显示。x 的位置为 1 到行宽范围内。因此使用 TAB 可以使输出项在一个输出行的任何位置上输出。

例 7.2.3

```
for i=1 to 3
  print tab (i); i
next i
for i=1 to 3
  print tab(i); i
```

```
next i
for i=1 to 3
  print tab(i); tab(i); i
next i
end
```

```
Ok. run
␣1␣
␣␣2␣
␣␣␣3␣
                                          1
                                          2
                                          3
␣1

␣␣2

␣␣␣3
Ok.
```

由上述程序中可以看出 TAB 定位的一些特点：

(1) 当 TAB(x)与输出项之间以分号分隔时，输出项按 x 指示的位置显示(是第一组输出)；

(2) TAB(x)与输出项之间以逗号分隔时，该 TAB(x)为无效(是第二组输出)，这时仅逗号起作用；

(3) 当打印的当前位置已超过 x 的位置时，输出项在下一行 x 指定的位置显示。如第三组输出，当 I=1 时，由于第一个逗号已使打印的当前位置移到第 2 区，再遇到 TAB(1)时，本应在第 1 列处开始输出 1，但已不可能，故移到下一行的第 1 列输出 I 的值。

此外，TAB(x)中的 x 可以用一个数值表达式给定。

例 7.2.4 设计一个打印如下图形的程序。

(列数)

(行数)	1	2	3	4	5
1	*				
2		*			
3			*		
4				*	
5					*

程序总共要打印五行。首先想到是否可以用一个循环结构

```
FOR N=1 TO 5
    打印一行“*”
NEXT N
```

进行打印。

下面考虑，在上述结构中如何“打印一行‘*’”。现作如下分析：

打印行数:1 2 3 4 5

打印位置:1 2 3 4 5

显然，每行的打印位置恰好等于行数。所以有可能用 Tab(N)来定义打印位置。即把“打印一

行‘*’”细化为

PRINT Tab(N); “*”

到此，程序便可以直接写出了。

```
for n=1 to 5
    print Tab(n); "*"
next n
end
```

7.2.3 自定义输出数据项格式(PRINT USING 语句)

在 PRINT 语句中使用逗号、分号或 TAB 函数，虽然可以使数据打印在指定的位置上，但是却不能指定一个数据本身的格式，例如，指定输出的数据值有几位小数；使上下二行的数按小数点对齐；在输出的一个数值中每三位数加一个逗号；使代表钱款的数值前后加一个“$”符号等等。

如果有以下语句：

```
for i=1 to 3
print name$(i), pay(i)
next i
 ⋮
```

打印出的格式如下：

```
zhang li 79.5
wang Fun 123.45
li Bin      89
```

可以看到：工资数据上下并不按小数点对齐，而是按左边第一个数字对齐。显然这对于应用(例如银行、统计部门)是很不方便的。

True BASIC 提供了 PRINT USING 语句，用来指定按什么格式输出**一个数据**。

PRINT USING 语句的格式为

PRINT USING〈格式串〉:〈数据项 1〉,〈数据项 2〉,……

或

PRINT USING〈格式串〉:〈数据项 1〉,〈数据项 2〉,……

“格式串”用来示范性的表示数据项所占的长度以及以什么样的格式输出，所以“格式串”也称为“示范串”。组成格式串的字符是系统预先指定的。每个格式字符隐含一定的格式含义。不同的 BASIC 版本所使用的格式字符集及其规定不同。True BASIC 中使用的格式字符集为

〈	〉	+	−	$	*	%	#	,	.	^

这些格式字符分别用于表示数值数据和字符串数据的格式。下面分别介绍它们的用法：

1. 数字格式字符:用以示范数值数据的格式字符有以下一些:

· 首部字符

\+ 在数字前面输出"+"或"-"。

\- 在数字前面输出空格(对正数)或"-"(对负数)。

$ 在数字前面输出"$"(在长城 0250 上是¥)。

· 数字型字符

* 将前置零用"*"代替。

% 将前置零输出"0"。

\# 将前置零用空格代替。

· 其它字符

, 将逗号插入在数字中适当的位置。

· 指定小数点的位置。

∧ 输出指数部分(按科学计数法)。

下面分别举例说明这些格式字符的用法。

(1) 简单数字格式

表 7.1

数	格式串	结 果	说 明
1	###	"␣␣1"	
-1	###	"-␣1"	负号在最左面
123	###	"123"	
-123	###	"***"	数据太长
5.67	###	"␣␣6"	四舍五入
-5.67	###	"-␣6"	
1	%%%	"001"	
-1	%%%	"-01"	
1	***	"**1"	
-1	***	"-*1"	

每一个"#"、"%"、"*"符号的位置,可以用一个数字字符代替。例如,示范格式串为"###"的含义:允许输出三个数字,如果数字个数不足三位,则前置零用空格代替。请注意,上述格式串没有小数点,因此它不能表示小数,只能表示整数,对小数按四舍五入处理成整数。

(2) 带小数的数字格式

如果想显示带小数点的数值数据,可在格式串中加一个小数点,输出时按指定的小数位数来截取。

表 7.2

数	格式串	结 果
1.1	###.###	"␣␣1.100"
-1.1	###.###	"-␣1.100"
333.666	###.###	"333 .666"
1.1	%%%.%%%	"001 .100"

由此看出：可以利用它来算输出的各行数据按小数点对齐。例如，

```
for i=1 to 3
    print using "###.##"; pay(i)
  next i
```

输出为

```
␣79.50
123.45
␣89.00
```

各行已按小数点对齐。

（3）起始符号格式

在格式串前面加入首部字符，可以在输出数字前面加上所需的符号。

表 7.3

数	格式串	结　果
63	+++#	“␣+63”
63	---#	“␣␣63”
-63	+++#	“␣-63”
-63	---#	“␣-63”
-63	-###	“-␣63”
63	$$$#	“␣$63”
-63	$$$#	“$-63”
63.2	$$$$$$#.##	“␣␣␣␣$63.20”
63.2	+++%%%%.%%	“␣␣+0063.20”
63.2	$$$$$+#.##	“␣␣␣$+63.20”
63.2	$******.##	“$****63.20”
63.2	+++++$$.##	“␣␣␣+$63.20”

（4）带逗号格式

在格式串中插入一些逗号，如果输出数值在逗号相应位置的两边都有数字，则在这些位置上输出逗号，否则逗号用空格代替。

表 7.4

数	格式串	结　果
63	##,###	“␣␣␣␣63”
2864	##,###	“␣2,864”
4e6	##,###,###,##	“␣␣␣40,000,00”
4321	$#,###,##.##	“$␣␣␣␣4,6326.00”

第一行的数据，由于 63 仅有两位数字，故不应在右边第三位数字前加逗号，逗号位置用空格代替，逗号可以加在任何所需的位置上。

(5) 科学表示法格式

表 7.5

数	格式串	结 果
4.5e3	###∧∧∧	"450e+1"
4.5e3	#,## ∧∧∧	"4,50e+3"
4.5e3	#,## ∧∧∧∧∧	"4,50e+003"
4.5e3	##,####∧∧∧	"45,0000e−2"
4.5e3	−#,####∧∧∧	"␣4,5000e+3"
−4.5e3	−#,####∧∧∧	"−4,5000e+3"
4e100	#,#∧∧∧	"******"

当格式串的尾部带有三个以上的符号"∧"时，便以科学表示法输出数值。指数部分的位数(包括"e+"或"e−")由"∧"的个数决定。

(6) 溢出格式

如果数字的整数部分比格式串所限定的整数部分的空间大，则输出一个星号串。

数	格式串	结 果
246	##	"* *"
−12	##	"* *"
543	$$#	"***"
12	##	"***"

2. 字符串格式字符

(1) 基本格式

字符串可以由任意数字型格式符号定义输出数据格式。即在*、#、%的相应位置上可以出现字符串中的一个字符。当输出的字符串短于格式串的长度时，多余的位置以空格形式分布在字符串两边，输出结果放在中间位置上(空格总数为奇数时，右边的空格比左边的空格多一个)；如果字符串比定义的格式串长，全部输出星号。

例 7.2.5

```
let a$="######"
let b$="%%%%%%"
let c$="$**.##"
print using a$ : "x"
print using b$ : "xy"
print using c$ : "xy"
print using a$ : "abcdefgh"
```

```
end
```

```
Ok. run
␣␣x␣␣␣
␣␣xy␣␣
␣␣xy␣␣
******
OK.
```

(2) 使字符串靠左端或靠右端

在格式串前加下列符号可以使字符串在所定义的空间内靠左端或靠右端。

- < 使字符串左对齐
- > 使字符串右对齐

例 7.2.6

```
let a$=">######"
let b$="<%%%%%%"
let c$="<$*、*##"
print using a$ : "x"
print using b$ : "xy"
print using c$ : "xy"
print using a$ : "abcdefgh"
end
```

```
Ok. run
␣␣␣␣␣x        (右对齐)
xy␣␣␣␣␣       (左对齐)
xy␣␣␣␣␣       (左对齐)
*******
Ok.
```

7.2.4 MAT PRINT USING 语句

MAT PRINT USING 语句是 MAT PRINT 语句和 PRINT USING 语句的组合。它用一个共同的格式输出一些数组的元素。其格式为

MAT PRINT USING〈格式串〉:〈数组 1〉,〈数组 2〉,……

当使用 MAT PRINT USING 语句时,数组中的每一行分别占一个输出行。如果有一个 2 行 3 列的数组,则输出为 2 行。如果用一条 MAT PRINT USING 语句显示几个数组,则在输出完一个数组后输出一空行。

例 7.2.7

```
! MAT PRINT USING.
!
```

```
dim a(3,4)
for i=1 to 3
   for j=1 to 4
      let a(i,j)=10*i+j                          ! assign values
   next j
next j
mat print using "####.# $##":a              ! print array
end
```

```
Ok. run
␣␣11.0␣$12␣␣13.0␣$14
␣␣21.0␣$22␣␣23.0␣$24
␣␣31.0␣$32␣␣33.0␣$34

Ok.
```

7.2.5 USING $ 函数

USING $ 函数可以把一些数值或字符串表达式的值转换成格式化的字符串。其格式为

USINT $ （<格式串>,<表达式 1>,<表达式 2>……）

函数中以一个格式字符串为第一变量。该格式字符串的定义方法与 PRINT USING 中的格式字符串相同。

USING 函数与 PRINT USING 语句的不同之处在于，它有可能把得到的格式化了的数据保存到一个字符串变量中供以后使用。

例 7.2.8

```
dim a$ (10)
for i=1 to 10
  input x
  let b$="####.##"
  let a$(i)=using$(b$,x)
next i
end
```

执行该程序后，便将所有输入的数值数据按 b $ 所定义的格式存放在数组 a $ 中。

习　题

7.1.1 写一个程序段检查下列问题的输入数据的合理性与合法性。

(1) 输入三个数据，求以这三个数据为三条边的三角形的面积。

(2) 求 C_m^n。

(3) 打印一份你班同学的简况登记表，内容包括每个同学的学号、姓名、性别、年龄、入

学成绩。

7.1.2 试举五个不好的程序输入风格的例子。

7.1.3 试比较 GET KEY 语句、KEY INPUT 函数与 INPUT 语句的区别。

7.2.1 试比较用 TAB 函数与用改变区宽的方法确定在下一行中的显示位置的方法有何异同?

7.2.2 指出下面程序的显示结果

```
let a$ ="1234567890"
let b$ ="ABCDEFGHIJ"
print TAB(10); a$ ; b$
print TAB(10); a$ & b$
end
```

7.2.3 用程序显示下列图形

(1)

```
V       V
 V     V
  V   V
   V V
    V
```

(2)

```
    ○○
   ○○○○
  ○○○○○○
 ○○○○○○○○
○○○○○○○○○○
```

(3)

```
    ×            ×            ×
   ×××          ×××          ×××
  ×××××        ×××××        ×××××
 ×××××××      ×××××××      ×××××××
×××××××××    ×××××××××    ×××××××××
 ×××××××      ×××××××      ×××××××
  ×××××        ×××××        ×××××
   ×××          ×××          ×××
    ×            ×            ×
```

7.2.4 某组有 6 人,设为 A,B,C,D,E,F,它们的外语成绩分别为 85,72,93,67,79,83,试编一程序,能按如下两种格式显示出该组中每个人的外语的成绩,总成绩(SUM)及平均成绩 AV。

(1) 格式 1

```
显示位置:  1    8    15    22    29    36
           A    B    C     D     E     F
           85   72   93    67    79    83
             空一行
           SUM=␣***
           AV=␣**.**
```

(2) 格式 2

```
显示位置:  1    8    15    22    29    36
           A    B    C     D     E     F
          ␣85  ␣75  ␣93   ␣67   ␣79   ␣83
             空一行
```

SUM=␣＊＊＊

AV=␣＊＊·＊＊

7.2.5 打印杨辉三角形："杨辉三角形"是我国南宋数学家杨谦光根据北宋数学家贾宪的"开方作法本源"归纳成便于记忆的，指数为正整数的二项式定理系数表(图 7.2 是五阶以内的二项式定理系数表)。它比欧洲的"帕斯卡三角形"早 600 年。

```
                    1
                 1     1
              1     2     1
           1     3     3     1
        1     4     6     4     1
     1     5    10    10     5     1
```

图 7.2

请设计一个能显示 N 阶杨辉三角形的 True BASIC 程序。

7.2.6 设计一个打印英文信封的程序。

7.2.7 按下面的要求编写一个口令检查程序。

(1) 完整的口令由 9 个字符组成；

(2) 这 9 个字符要分三次口答；

(3) 每周内星期一到星期日七天中的回答均不同(口令本身不变)。

第八章 图 形

图形是一种重要的信息表示形式。与文字信息相比，图形具有容易理解和记忆，能容纳大量信息等特点。目前，计算机应用的多种领域，如计算机辅助设计(CAD)、计算机辅助制造(CAM)、计算机模拟、图像处理、地学信息处理及地图绘制、信息系统中的数据分析与辅助决策、计算机辅助教学(CAI)、游戏等方面，都要求计算机输出图形。

True BASIC 有很强的图形功能。它的绘图语句简单、直观；可以方便地画点、线、弧、圆、方、实心圆与实心方；可以进行图的着色、放大、缩小、旋转，为图形附加正文；可以定义图的输入、实现动画技术，定义与子程序等价的图画。

本章简要介绍 True BASIC 图形处理的基本功能。

8.1 显示模式与图形窗口

8.1.1 显示器的工作模式

IBM PC 可以配置“单色”CRT 或具有“彩色/图形适配器”的 CRT，并且可以有正文与图形两种基本工作模式(mode)。

在只配有“单色”CRT 的机器，只能在单色正文模式下工作。配有“彩色/图形适配器”的机器，不仅可以在正文显示模式下工作，还可以在图形显示模式下工作。

文本显示模式也称字母/数字模式(A/N 模式)，它显示的是字符。通常每行可显示 80 或 40 个字符。在彩色显示器上可显示彩色字符。

在图形显示模式(APA 模式)下，可以由程序直接控制称为“像元”(pitel)的屏幕上的基本单位，一个“像元”就是通常所说的一个“点”。每个点的颜色和亮度可以分别进行控制。一幅图画实际上是由成千上万个“点”组成的。APA 模式又可分为三种不同的分辨率：

(1) 高分辨率模式：采用 200×640 点阵。

(2) 中分辨率模式：采用 200×320 点阵。

(3) 低分辨率模式：采用 100×160 点阵。

在 IBM PC 机上，True BASIC 图形功能要在中分辨率图形模式下才能使用。在高分辨率模式下虽然也可以作图，但只能使用一种颜色。

8.1.2 图形窗口坐标

1. 设置图形窗口坐标：人们平时作图习惯于在一个平面坐标系中进行。而显示屏上都以“像元”为单位排列地址。显然，如果要用户在程序中直接指定哪个“像元”具有什么颜色、什么亮度是很不方便的。True BASIC 允许用户按照坐标位置定义图形的各个点，而不必考虑它们对应于显示屏上哪些像元点。当用户给出图形中某一个点的坐标时，True BASIC 会

自动把它转换为显示屏的像元地址。

然而，计算机的屏幕是不可能装下整个坐标系的(坐标系是无限的)，它只能显示坐标系中的一个矩形局部，这个矩形称为平面坐标系中的一个图形窗口。计算机屏幕上的图形就是这个窗口的映射，绘制在窗口以外的点在窗口边上被"修剪"掉了。如图 8.1 所示。屏幕所显示的是坐标系中(1900,0),(1980,0),(1980,150),(1900,150)四个点组成的矩形窗口。True BASIC 通常在图形窗口输出图形。使用图形窗口时编辑与背景窗口便全消失。

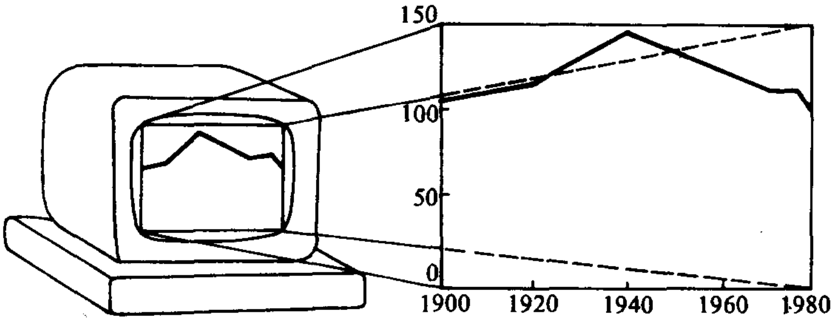


图 8.1

因此使用 True BASIC 作图的第一步是设置窗口坐标。设置窗口坐标的语句是 SET WINDOW。

设平面坐标系的水平方向为 X 轴，垂直方向为 Y 轴，则 SET WINDOW 语句的一般形式是

SET WINDOW Xmin, Xmax, Ymin, Ymax

这里 Xmin 为图形窗口的左端坐标，Xmax 为其右端坐标，Ymin 为其底边坐标，Ymax 为其顶边坐标。如图 8.1 中的图形窗口坐标用下面的语句建立

SET WINDOW 1900,1980,0,150

如果 Ymin>Ymax 则图形将被上下颠倒。同样，若 Xmin>Xmax，则图形将会被左右翻转。

如果用户不定义窗口坐标，True BASIC 会自动将窗口定义为(0,0)到(1,1)。然而，这个窗口并非适合所有的图形。所以在绘制任何图形之前，都应当先执行一条 SET WINDOW 语句建立所需的窗口坐标。这样程序的可读性也比较好。

需要时，用户还可以用 ASK WINDOW 语句来查询当前窗口坐标。ASK WINDOW 语句的格式是

ASK WINDOW Xmin, Xmax, Ymin, Ymax

其中 Xmin, Xmax, Ymin, Ymax 为四个变量名，可以用 PRINT 语句输出它们的值。

2. 纵横比：在作图时，会注意到这样一个事实：在坐标系统中画一个正方形，但在显示器的屏幕上显示时往往变成了矩形，这是由于屏幕是矩形的，其纵横比不是 1∶1。如果定义窗口为边长为 10 的正方形，在屏幕上却用一个矩形来反映它。换句话说，X 轴上的一个单位，要比 Y 轴上的一个单位要长些。因此，一个正方形的窗口，映射到屏幕上时，窗口中的圆

便会变成椭圆，如图 8.2(a)所示。定义窗口为正方形，在窗口内画一个内切圆，在屏幕图形显示出一个椭圆图形。为了在屏幕上得到圆，必须调整窗口的纵横比，即使坐标系统的窗口定义为矩形，屏幕也是矩形，二者的纵横比一致。如图 8.2(b)所示。

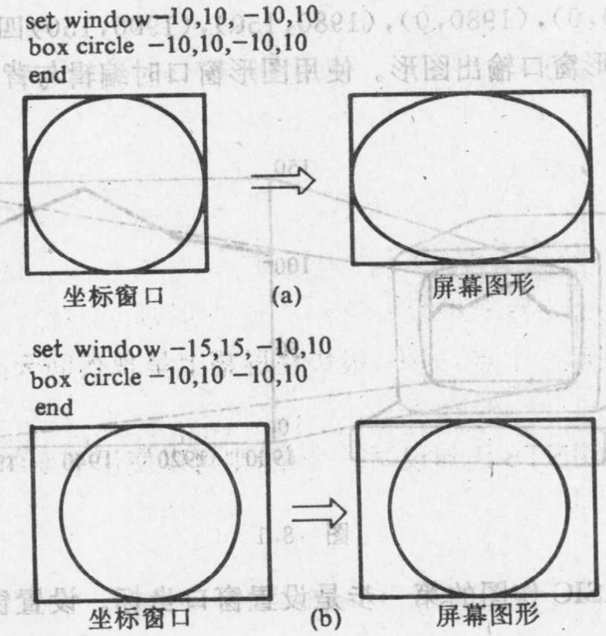


图 8.2

由于各显示器的屏幕的纵横比不同，因此在某种机器上的画圆程序，在另一种机器上执行时，可能还须重新调整纵横比。

3. 多重窗口：前面介绍的窗口坐标被称为逻辑坐标，象原点坐标称为物理坐标。程序员使用 True BASIC 作图时，所使用的是逻辑坐标。逻辑坐标与物理坐标之间的转换是由 True BASIC 自动进行的。如果需要，可以把几个不同窗口的图形映射到同一屏幕上，组成一幅新的图形。

4. 清除图形窗口：True BASIC 用 CLEAR 语句清除当前图形窗口，以便重新画另一张图形。

8.2 画 图

8.2.1 画点

显示器屏幕上的每一幅画面都是由成千上万个的点(像元)组成的。换句话说，为了在屏幕上产生一幅图形，首先要求能在屏幕上的某一位置画一个点。

True BASIC 使用 PLOT POINTS 语句画点。其格式为

```
PLOT POINTS：X1,Y1,X2,Y2,……
```

其中，每个(X,Y)描述了当前图形窗口坐标中一个点的位置。如果某个点在当前窗口之外，则它不在屏幕上显示。

例 8.2.1 繁星密布。

```
! Plot random points, like stars
!
do
  let x=Rnd                         ! Pick two random numbers
  let y=Rnd
  plot points : x,y                 ! Plot a point at the random spot
loop until key input
end
```

上面的程序使用缺省窗口,即图形窗口坐标为 0,1,0,1。而由两个 Rnd 所产生的点的坐标(X,Y)也在此区间内。故程序开始运行后,首先在屏幕上的某一随机位置显示一个点,接着又在另一随机位置显示一个点,……,很快在屏幕上呈现夜间天空中繁星密布的景象,直到按下任一键为止。

如果一个 PLOT POINTS 语句仅画一个点,则可以使用下面的简略形式:

```
PLOT X,Y
```

8.2.2 画线

True BASIC 提供的画线语句为

```
PLOT LINES : X1,Y1;X2,Y2;……
```

或

```
PLOT X1,Y1;X2,Y2;……
```

其中分号表示从前一个坐标点向后一个坐标点引一直线。如 plot 0,0; 0,1 表示由坐标点(0,0)向(0,1)引一直线。而

plot 0,0;0,1;1,1;1,0;0,0

表示画一个单位正方形。

当 PLOT 语句以分号结尾时,表示其最后一个坐标点,将向下一个 PLOT 语句中的第一个坐标点引一直线。如可以用下面的程序段来画一个单位正方形。

```
plot lines:0,0;0,1;
plot lines:1,1;
plot lines:1,0;0,0
```

例 8.2.2 画一条正弦曲线。程序如下:

```
! plots a sine curve
!
set window 0,2 * Pi, -1, 1
for x=0 to 2 * Pi step.1
  plot x, Sin(x);
next x
```

end

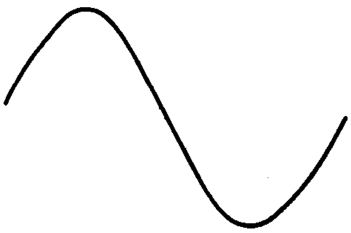


图 8.3

请注意循环中的语句:plot x,sin(x);它的作用是:当 x=x1 时,从点(x1,sin(x1))向下一点(x2,sin(x2))作一直线,再由第二个点向第三个点(x3,sin(x3))作一直线。x 的值由 0 变到 2π,每次变化 0.1 弧度。画出的一条条小直线形成如图 8.3 所示的正弦曲线。

8.2.3 画矩形

用画线的方法可以画出矩形框来,但它的执行速度较慢。True BASIC 提供了一个专用的画矩形的语句,可以快速画出矩形框,其句型为

```
BOX LINES xmin, xmax, ymin, ymax
```

矩形框中各边的坐标位置是以窗口坐标为基准的。

例 8.2.3

```
! Draw a box
!
SET WINDOW −14,14, −10,10
! 0,0 is center of window
BOX LINES −5,5, −5,5
! 5 * 2 is length of eash side
END
```

程序画出的方框如图 8.4 所示。

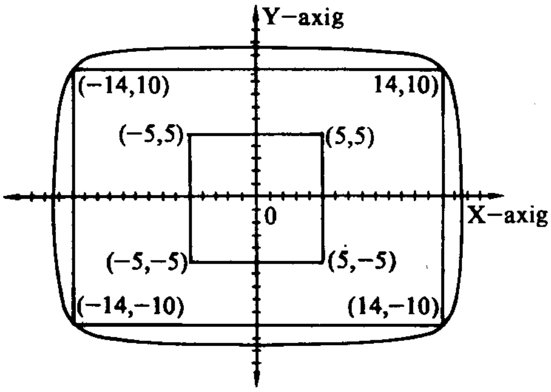


图 8.4

8.2.4 画圆与椭圆

BOX 语句还可以用来划圆或椭圆,句型为

```
BOX CIRCLE xmin, xmax, ymin, ymax
```

或

```
BOX ELLIPSE xmin, xmax, ymin, ymax
```

这两种句型通用。当"盒子"是正方形时,画出圆,当"盒子"是长方形时,画出椭圆。如图 8.5 所示。

语句(a):

BOX CIRCLE −5,5, −5,5

或 BOX ELLIPSE −5,5,−5,5

语句(b):

BOX CIRCLE −8,8, −5,5

或 BOX ELLIPSE −8,8,−5,5

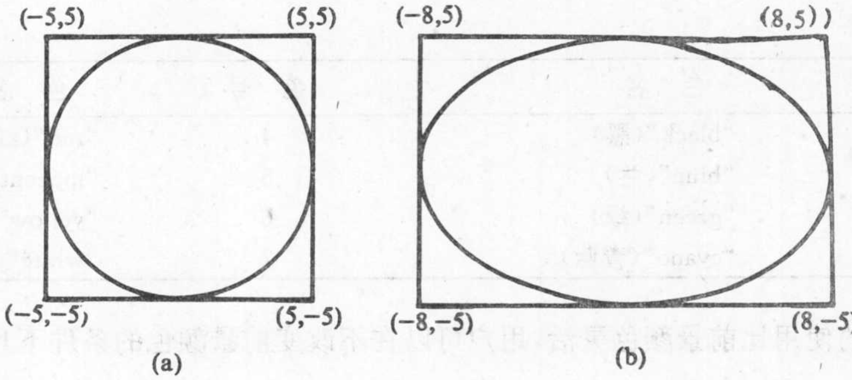


图 8.5

8.3 着 色

8.3.1 前景颜色与背景颜色

背景颜色是指没有任何显示时的屏幕颜色，也称为底色。前景颜色是所显示的字符或所绘制的图形的颜色。8.2 节中介绍过的画图语句所画的图都是用绿色作为前景颜色，以黑色作为背景颜色。要使用其它颜色，要先设置前景颜色和背景颜色。

设置前景颜色的 True BASIC 语句为 SET COLOR。其句型为：

SET COLOR"〈色名〉"

或

SET COLOR〈色号〉

在 IBM PC 上使用的前景色如表 8.1 所示。

表 8.1

色 号	色 名	色 号	色 名
0	"background"(背景色)	4	"cyan"(青兰)
1	"green"(绿)	5	"magenta"(洋红)
2	"red"(红)	6	"white"(白)
3	"yellow"(黄)		

按正常的图形模式，IBM PC 上可同时显示出三种前景颜色。能同时使用的颜色分为两组，称为两种模式：红、绿为一组，青兰、洋红、白为另一组。在同一时刻仅能使用一种模式中的颜色。如已使用红色作为前景颜色作图了，现在又要用白色作为前景颜色绘一些图，原来的红色将自动转换成洋红色。

设置背景色的语句有如下两个等价形式：

SET BACK "〈色名〉"(或〈色号〉)

SET BACKGROUND COLOR "〈色名〉"(或〈色号〉)

在 IBM PC 上使用的背景色如表 8.2 所示。

表 8.2

色 号	色 名	色 号	色 名
0	"black"(黑)	4	"red"(红)
1	"blue"(兰)	5	"magenta"(洋红)
2	"green"(绿)	6	"yellow"(黄)
3	"cyano"(青蓝)	7	"white"(白)

背景颜色的使用比前景颜色灵活,用户可以在不改变前景颜色的条件下,使用各种不同的背景颜色。

如果要想同时设置前景与背景色,可以使用语句

SET COLOR "〈前景色名〉"/"〈背景色名〉"

在程序中,还可以用 ASK 语句来查询当前色名或色号。

查询前景颜色的 ASK 语句格式为

ASK COLOR 〈串变量〉

或 ASK COLOR 〈数值变量〉

查询背景颜色的 ASK 语句格式为

ASK BACKGROUND COLOR 〈串变量〉

或 ASK BACKGROUND COLOR 〈数值变量〉

这两个语句也可以简化为

ASK BACK 〈串变量〉

或 ASK BACK 〈数值变量〉

执行 ASK 语句的结果是将代表当前颜色的一个大写字母赋给指定的串变量或将色号(数字)赋给指定的一个数值变量。

应当注意的是,每种机器上可以使用的颜色是不同的。如有的允许使用几千种不同的颜色和深浅程度,而有的只允许使用两种(黑、白)颜色。为了知道你所用的计算机上所能显示的前景颜色的最大数目,可以使用下面的 ASK 语句查询:

ASK MAX COLOR 〈数值变量〉

它能将在同一时间内,在显示屏幕上显示的颜色种类的最大数目赋给指定的数值变量。

你所能使用的颜色代码便是 0～指定的数值变量的值之间的数值。"0"代表背景色。如果使用了超出这个范围的数值的色号,True BASIC 会给出一种合理的替代颜色。至于各个数值所代表的颜色是什么,请查阅你所使用的计算机用户指南。

8.3.2 用 BOX AREA 语句画实体矩形

用 BOX AREA 语句代替 BOX LINES 语句，可以画出一个用当前颜色涂满的矩形块。

8.3.3 用 PLOT AREA 语句画实体图形

用 PLOT AREA 语句可以画一个形状复杂的实体图。PLOT AREA 语句的格式为

```
PLOT AREA:x1,y1;x2,y2;……
```

执行 PLOT AREA 语句时，光束先从点(x_1，y_1)画一条线到点(x_2，y_2)，再从点(x_2，y_2)画一条线到点(x_3，y_3)，……，最后画一条线返回点(x_1，y_1)，把区域封闭起来。接着便用当前色填满这个区域。

8.3.4 用 FLOOD 语句着色

FLOOD 语句的格式为

```
FLOOD x,y
```

它的功能是将点(x，y)的颜色扩散到具有相同颜色的邻近的像元点上。常常用于：先改变某一点的颜色，然后用 FLOOD 语句将此颜色扩散到与该点原来颜色相同的那些点上去。

例 8.3.1

```
! Bull's eye pattern
!
set window -14,14,-10,10          ! 0,0 is center of window
for i=10 to 1 step -1             ! From large to small
    set color Mod(i,2)+1          ! Alternate green & red
    box circle -i,i,-i,i          ! i is radius of circle
    flood 0,0                     ! Fill-in from center
next i
end
```

运行该程序，可以产生由大到小 10 个绿、红交替的实心颜色同心圆。

8.3.5 图形中的正文设置

True BASIC 提供了一条 PLOT TEXT 语句，它的句型为

```
PLOT TEXT,AT x,y:〈字符串表达式〉
```

它的作用是把字符串绘制在图像上坐标(x，y)的右上方。它的输出格式同 PRINT 语句(也可以使用 USING $ 函数等)。

例 8.3.2

```
! PLOT TEXT with histogram
```

```
!
set window 0,6,0,150
for x=1 to 5
    read value
    box area x- .4,x+ .4,0, value
    plot text,at x-.4,value +10:using $ ("#
    ##・#", value)
next x
data 20.35, 41.77, 96.33, 65.2, 23.93
end
```

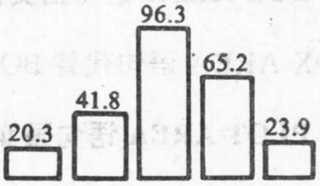


图 8.6 带有标号的直方图

执行该程序将画出如图 8.6 所示的直方图。

8.4 动 画

动画是指屏幕上显示出来的画面(或画面中的一部分),能够按一定的规则在屏幕上作相对运动。它能使计算机显示的输出图形具有更强的真实感,在CAI、游戏、模拟等应用中十分有用。

动画是对动画对象重复地进行显示——变换——擦除——再显示的结果。在 True BASIC 中,实现简单动画的方法是,先把屏幕上某一矩形平面中的图像,记录到一个字符串变量中,接着反复进行以下操作:删除矩形——在一定距离的位置上重现——再删除——再重现。这些功能都是用 BOX 语句实现的。

1. 保留当前"盒子"的常数于一字符串变量中,句型为

BOX KEEP xmin,xmax,ymin,ymax IN〈字符串变量〉

2. 删除"盒子",句型为

BOX CLEAR xmin,xmax,ymin,ymax

3. 再现"盒子",句型为

BOX SHOW〈字符串变量〉AT xmin,ymin

它给出的 xmin,ymin 为盒子再现时左下角的坐标位置。

8.5 图 画

True BASIC 提供的图画实际上是子程序的一种形式。在定义一个图画后,可以根据需要随时调用它,在调用时还可以作移位、改变宽度、放大缩小、旋转、倾斜等五种变换。与子程序一样,图画可以是内部的(定义在 END 之前)或外部的(定义在 END 之后),也能放在库文件中。

8.5.1 图画的定义与调用

一个典型的图画结构如下：

```
PICTURE 〈图画名〉(〈变量 1〉,〈变量〉,……)
    ———
    ———  }语句块
    ———
    PLOT……
    ———
    ———  }语句块
    ———
END PICTURE
```

图画由 DRAW 语句调用，既可以嵌套调用，也可以递归调用，DRAW 语句的句型有以下几种：

(1) DRAW〈图画名〉

(2) DRAW〈图画名〉WITH〈变换〉

(3) DRAW〈图画名〉WITH〈变换$_1$〉*〈变换$_2$〉

8.5.2 图画的变换

True BASIC 能为一个 DRAW 语句提供表 8.3 所示的五种变换。

表 8.3

变换名	意 义	变换名	意 义
Shift(a,b)	移动(x,y)到(x+a,y+b)	Rotote(a)	围绕原点按逆时针方向将图旋转 a 弧度
Scale(a,b)	改变(x,y)为(x*a,y*b)	Shear(a)	将垂直线向右倾斜 a 弧度
Scale(a)	改变(x,y)为(x*a,y*a)		

当在 DRAW 语句中使用〈变换$_1$〉*〈变换$_2$〉时，是一种复合变换，可以同时进行两种变换。

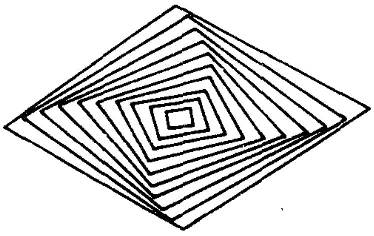


图 8.7 旋转的正方形

例 8.5.1

```
! Draw Squares tipped on an angle
!                connect four corners
Picture Square(size)
```

```
    plot -size,size;
    plot size size;
    plot size,-size;
    plot -size,-size;
    plot -size,size
  end picture
  set window -15,15,-15,15                     ! define window coordinates
  for  i=1 to 10                               ! use sizes 1 through 10
       draw Square (i) with Rotate (i * Pi/40) ! draw square of this size
  next i
  end
```

执行该程序可以得到图8.7所示的图形。

以上只对 True BASIC 的画图功能作了简单介绍，如果读者需要更详细的了解，可参考 True BASIC 使用手册。

习　　题

8.1 画一个五角星。

8.2 画一个会闪烁的五角星。

8.3 画一个球作圆周运动。

8.4 画一个五个班平均成绩的直方图，五个班的成绩顺序为：86.5，72.5，88，97，63。要求在直方形下面用文字注明"1班"，"2班"，……，在直方形上方注出分数。

8.5 请你设计一幅汽车向前行进的动画。

8.6 请画出一个球以45°角度向上斜抛后的运动轨迹。向上抛的初速 $v_0=100\text{cm/s}$。计算公式：

$$x=(v_0\cos45°)\cdot t$$

$$y=(v_0\sin45°)t-\frac{1}{2}gt^2$$

其中 t 是时间，g 是重力加速度，$g=981\text{cm/s}^2$。

第九章 数据文件

计算机的内存昂贵而容量较小。并且，由于它是由电子元件做成的有源存储元件，一旦停电或关机，内存中的信息便会丢失。电子计算机的外存价廉而容量可以做得很大，象磁盘与磁带可以不断更换，其容量可以看成是无限大的。并且，外存是用磁表面介质材料(其原理象盒式录音带)做成的无源存储元件，所存储的信息不会因停电而丢失。因此常常用外存保存暂时不与 CPU 交换的一些信息。

信息在外存上以文件的形式被组织在一起。文件是一组具有一个名字的相关信息。这个名字称为文件名。在用软磁盘做外存的系统中，一片软盘可以保存多个文件。文件按其内容可分为程序文件与数据文件两大类。程序文件所保存的内容是程序。关于程序文件的操作已在2.1.4节中作了概要介绍。这一章主要介绍数据文件的有关操作。数据文件所保存的内容是数据。

9.1 有关概念

9.1.1 文件名

文件名是它所标识的文件中所包含信息的总称。当对外存中的数据进行操作时，首先要指定其所在文件的文件名。以便能让计算机的操作系统在众多的文件中找到存储要操作数据的文件，协助用户程序完成存取操作。

文件名是按一定的格式和规则组成的。由于文件操作是在操作系统支持下进行的，文件的命名规则应符合操作系统的要求。True BASIC 是在2.0版本以上的 DOS 的支持下运行的，它的文件名应遵守 DOS 2.0对文件名的命名要求。这些规则已在2.1.4节中介绍，这里不再赘述。但须指出的是，当要对非当前驱动器盘中的文件进行操作时，还应在其文件名前标以所在的驱动器号。如当前驱动器为 A，而要对插在 B 驱动器中的磁盘上的文件 myfile.aaa 进行操作时，应标识为

B:myfile.aaa

对插在当前驱动器中的磁盘上的文件进行操作时，可以标上当前驱动器的标号，也可以不标。

9.1.2 文件的存取方式与文件指针

文件按其对元素的存取是否一定要按顺序进行，可分为顺序文件与随机文件两种。对顺序文件中的元素进行读或写时，只能按顺序依次地进行，即只有读或写完第 i 个元素，才能读或写第 i+1个元素。随机文件则没有这种限制。它允许通过记录号等对任意一个元素(记录)进行读或写。

实际上，对文件元素的存取是按文件指针的指示进行的。文件指针是由操作系统设置的一种机制，它指向当前文件元素的位置。对该文件元素进行读或写后，指针便会依次自动指向下一个文件元素。所以说，不论顺序文件还是随机文件，它们的自然存取方式都是顺序的。

True BASIC 允许用语句 SET 或 RESET 移动指针的位置。但是对顺序文件来说，只能使指针移到文件首(begin)或文件尾(end)；而对随机文件来说，既可以将文件指针移到文件首、文件尾，还可以将文件指针移到下一个(next)元素、当前(same)元素，甚至任何位置。这就是随机文件与顺序文件的区别。关于移动文件指针的语句的具体用法，我们将在本章随后三节中介绍。

9.1.3 文件的组织(organization)

组织模式反映了文件的组织特征。按组织模式，True BASIC 数据文件可以分为：正文(TEXT)文件、记录(RECORD)文件和字节(BYTE)文件三类。这三类文件的特征区别如表9.1所示。

表 9.1

文件类型	组织模式	存取元素				存取方式
		元素名称	长度	组成成份	元素间分隔	
TEXT	一系列可显示的不定长数据行	字符行	不定长	一组数值、字符半数据组成的字符行	回车、换行	顺序
RECORD	一系列固定长度的记录	记录	定长	数值数据或字符串数据	无	随机
BYTE	一串结构上无限定的字节流	字节段(字符串)	可任意长	字节(字符)	无	顺序

例 9.1.1 TEXT 文件示例1：

```
! Program↙
LET a=5↙
FOR i=1 TO 10 ↙
        PRINT a+a;↙
NEXT i↙
END↙
```

这里每个程序行都是一个可显示行，也是 TEXT 文件中的一行。行间以回车换行为分隔符。

例 9.1.2 TEXT 文件示例2

```
X=␣2          y=␣3          x+y=␣5
```

上述 TEXT 文件中的字符行由字符串“X=”，数值数据“2”，字符串“y=”，数值数据

“3”，字符串“x＋y＝”，数值数据“5”组成。

例 9.1.3 一个 RECORD 文件的例子。

记录号	文件内容		指针位置
1	张 三		begin
2	50		
3	313.50		
4	李 四	←	same
5	52		next
6	258.60		
7	王 五		
8	47		
9	168.20		end

这里每个记录分别由一个姓名(字符串)或年龄、工资(数值数据)组成。

例 9.1.4 一个 BYTE 文件的例子。

ZHANG 350 313.50 LI452 258.60 WANG547 168.2

一个数字节流。

文件所占用的字节数称为文件的大小或长度。对记录文件，有时也依其记录数作为文件大小的一种尺度。

9.1.4 通道、文件的打开与关闭

通道是连接内存中的程序与外存中的文件的纽带。当内存中的程序要对某一文件进行数据读写时，必须启动相应的通道。这时也称该文件被打开了。

一个 True BASIC 程序最多可以同时打开10个数据文件。在对数据文件进行操作前，必须先打开文件，即给文件分配一个通道。True BASIC 程序中用语句 OPEN 执行打开文件的操作。OPEN 语句的基本格式为

OPEN 〈通道号〉:name“〈文件名〉”

执行该语句后，便给文件分配了一个通道。如语句

OPEN ＃37:name "ABCD. $"

执行后，将37号通道分配给文件“ABCD. $”，向37号通道写的信息将由操作系统写到文件“ABCD. $”中。

True BASIC 规定，通道号用[1，999]之间的整数表达式表示。

为整齐划一，True BASIC 也把其它外设当作文件进行管理，并把通道号＃0作为键盘与屏幕的固定通道。＃0通道总是打开的。

打开文件的另一个作用是把存于外存的及需说明的文件属性传送到内存。这些属性包

括：

(1) 文件名；

(2) 文件的访问约束：

input (只读)

output(只写)

outin(读写均可)

(3) 文件的长度、存储位置；

(4) 文件状态：

new (新文件，即要建立的文件)

old (旧文件，即已有的文件)

newold(不明)

(5) 文件的组织模式：

TEXT (正文)文件

RECORD(记录)文件

BYTE(字节)文件

(6) 文件元素的长度。

其中需说明的信息(1)，(2)，(4)，(5)将作为可选项附加在 OPEN 语句中。格式为在 OPEN 语句的基本格式后附加项

[,access a $][, create c $]

[, organization 0 $][, recsize expr2]

其中：

a $ 可取下列值之一：

input, output, outin(缺省值)

c $ 可取下列值之一：

new, old, newold(缺省值)

0 $ 可取下列值之一：

text, record, byte

expr2用以指定文件元素(text 文件中的行，record 文件中的记录，byte 文件中的字节段)的长度。取整数值。

例如语句

OPEN #5:name "abcdefgh. mmm", access input, organization record, recsize 40

执行后，将为当前盘上的文件“abcdefgh. mmm”分配一个5号通道，并且指明该文件为每个记录长度为40个字节的记录文件，此次打开后只执行读操作。

文件读写结束后，应及时将它所占用的通道释放。这时，操作系统将会把因读写造成的文件属性的改变内容写到磁盘中，以便下次使用。这一过程称为关闭文件。True BASIC 采用 CLOSE 语句关闭文件，其格式为

```
CLOSE #〈通道号〉
```

例如语句

CLOSE #5

将把前面执行打开操作时打开的5号通道关闭。

9.1.5 数据文件操作的一般过程

无论任何数据文件,尽管操作细节不同,都应遵照如下的基本操作过程进行:

s1:打开文件;

s2:文件读写;

s3:关闭文件。

也就是说,在文件读写之前必须先打开文件,在文件读写结束后应关闭文件。另外在文件读写时还应注意:

(1) 文件的读、写操作方式必须与打开时的设定相一致。

(2) 在每次读文件时,应先测试文件是否已经结束。即要测试当前文件指针处还有无数据或测试当前文件指针是否已指向了文件尾。前者可用函数

```
MORE #〈通道号〉
```

后者可用函数

```
END #〈通道号〉
```

当连续地进行读文件操作时,建议采用如下的循环结构:

```
DO WHILE more #〈通道号〉
    读操作
LOOP
```

9.2 正文文件操作

正文文件也可以称为行文文件。它与显示器屏幕上所显示的字符行具有相同的组成、格式,即它也是由一行一行可图示的字符组成。

9.2.1 写正文文件

1. PRINT 语句的一般形式

我们对 PRINT 语句已经很熟悉了。其实它是一个写正文文件语句,写屏幕只是它的一种应用。其一般形式为

```
PRINT   #〈通道号〉:〈表达式表〉
```

其中,"表达式表"由一列用","或";"分隔的数据项(数值或字符串)组成。","与";"的分隔功能如大家已熟知的那样。

例 9.2.1 简单的写正文文件程序。

```
!
! example 9. 2. 1
!
OPEN #3:name "ex921. txt", create new, access outin, organization text
LET x=3
LET y=5
PRINT #3:"x=";3,"y=";5,
PRINT #3:"x+y=";x+y
CLOSE #3
END
```

执行这一程序后,将在当前盘中建立一个名为"ex921. txt"的正文文件,并且在其中写了一行数据:

```
x=␣3            y=␣5            x+y=␣8
```

为什么两条 PRINT 语句只写了一行呢?这是因为第一条 PRINT 语句的最后有一个","的缘故。这个","既有按区分隔的作用,又有连接下一个打印语句的功能。

执行了两条 PRINT 语句,即写了一行正文文件(有时也称一条正文文件记录)后,文件指针指向"2",等待写入第2行。即每写一行,文件指针移动一次,直到文件被关闭为止。

2. 正文文件的行宽与区宽

PRINT 语句向正文文件中写数据是在一定的行宽中进行的。行宽可以用语句

```
SET #<通道号>:margin<行宽表达式>
```

设置。设置 MARGIN 为"maximum"(机器最大值)时,可以按任意长度输出各行。

","的功能是按区分隔各数据项的宽度。区宽可以用语句

```
SET #<通道号>:zonewidth<区宽表达式>
```

设置。

系统内定(隐含)值为行宽80,区宽16。

一个数据文件是可以被不同的程序进行操作的。当一个程序要对存在外存中的一个正文文件进行操作的情况下,必要时可以用下面的语句来询问该正文文件的行宽与区宽:

```
ASK #<通道号>:margin<变量1>
```

```
ASK #<通道号>:zanewidth<变量2>
```

执行这两个语句后,将分别把询问到的行宽(margin)与区宽(zonewidth)分别存到变量1与变量2中,以供别的语句使用。

3. 正文文件的重写

正文文件不可重写,企图重写一个已存在的正文文件时,系统将会发出

```
File already exists.
```

的警告。为了重写正文文件，应先用 ERASE 语句将文件内容擦除。但是允许向已存在的正文文件中追加数据。要向已存在的正文文件中追加数据，必须先置文件指针于“文件尾”(关闭文件时给出的标志)。因为文件的打开语句总是将文件指针置于文件开始，即第一行处。置文件指针于文件尾的语句为：

```
SET #〈通道号〉:POINTER end
```

或

```
RESET #〈通道号〉: end
```

4. PRINT 语句的其它形式

```
PRINT #〈通道号〉,USING:〈格式串〉:〈数据表〉
```

```
MAT PRINT #〈通道号〉:USING〈格式串〉:〈数据表〉
```

这两种句型的用法请分别参阅7.2.3、7.2.4节。

9.2.2 写屏幕与打印机输出

如前所述，True BASIC 把所有 I/O 设备都当作文件进行读写，并给屏幕与键盘以固定的通道号——#0。并允许使用 PRINT #0:〈表达式表〉的通道号缺省形式

```
PRINT〈表达式表〉
```

同时，由于0通道总是打开的，所以不需要打开与关闭语句。这可以说是前面所用的 PRINT 语句格式的来历。

True BASIC 给打印机一个固定的文件名:“Prn”。因此，要在打印机上进行输出应使用如下语句结构

```
OPEN #〈通道号〉:printer
PRINT #〈通道号〉:〈数据表〉
……
CLOSE #〈通道号〉
```

9.2.3 读正文文件

与 PRINT 语句相似，INPUT 语句其实是一个读正文文件语句。其基本句型为：

```
INPUT #〈通道号〉:〈变量表〉
```

它的功能是从打开的正文文件中读数据，而且是给每个变量读入一行。但是它不能读入空行，即用

PRINT #〈通道号〉

语句写入正文文件的空行。要用 INPUT 语句读入空行时，系统将给出警告：

```
Too few input items.
```

这时，应使用下面的句型

```
LINE INPUT #〈通道号〉:〈变量表〉
```

它能将任何形式的行读入内存。

例 9.2.2

```
!
! example 9.2.2
!
OPEN #3:name "ex922.txt",create new, organization text
LET x=3
LET y=5
PRINT #3                        !写一空行
PRINT #3:"x=3";3,"y=";5,
PRINT #3:"x+y=";x+y
IF more #3 then INPUT #3:a $    !读出空行
IF more #3 then INPUT #3:b $
PRINT a $
PRINT b $
CLOSE #3
END
```

执行该程序后，将在正文文件"ex922.txt"中写入两行。其中第一行是空行。同时在屏幕上显示出与 ex922.txt 中相同的内容。

```
x=3          y=5          x+y=8
```

其上述两种 INPUT 句型外，还可以使用：

```
MAT INPUT #〈通道号〉:〈变量表〉
```

```
LINE MAT INPUT #〈通道号〉:〈变量表〉
```

这些句型的用法请参阅前面的有关章节。

还须指出，正文文件虽不可重写，但可以重读。要重读时，应先使用下面的句型将文件指针置于文件首。

```
SET #〈通道号〉:POINTER begin
```

或

```
RESET #〈通道号〉:begin
```

9.2.4 按数据项处理正文文件

如前所述，一般情况下只能按行对正文文件进行读出。为了对含有多数据项的行进行处

理，可以采用如下两种办法。

(1) 用子字符串取出行内所含的各项。

(2) 在行中各数据项之间用“,”进行“硬”分隔。

例 9.2.3

```
!
! example 9.2.3
!
OPEN #3:name "ex923.txt", create new, organization text
LET x=3
lET y=5
PRINT #3:"x=";",";3;",";"y=";",";5;","
PRINT #3:"x+y=";",";x+y
CLOSE #3
OPEN #3:name "ex923.txt", access input
IF more #3 then INPUT #3:a$,b,c$,d,f$,g
PRINT a$;b,c$;d,f$;g
CLOSE #3
END
```

执行该程序后将在文件“ex923.txt”中写入

```
x=,3,    y=,5,    x+y=,8
```

并在屏幕上显示出

```
x=3         y=5         x+y=8
```

这时，应当注意

(1) 一条 INPUT 语句中的变量数应与所欲读的行中的数据项数一致，否则系统将发出警告：

```
Too many input items.
```

即不可用几个 INPUT 语句读一行中的数据。

(2) INPUT 语句中的各变量的类型应与所读行中各数据项的类型一致。

9.3 记录文件操作

9.3.1 True BASIC 记录文件的特点及其属性设置

1. 记录文件的结构特征

True BASIC 记录文件与一维数组有点相似，每个记录象一个数组元素一样只可以存储一个数据项，不同之处在于，数组是内存数据结构，且每个数组只能由同类型的数据组成，

而记录文件是外存数据结构且各个记录所存储的数据类型可以不相同。

记录文件一律以非 ASCII 码存储，无法直接在屏幕上直接显示所存储的实际内容。

2. 记录长度的设置与询问

正文文件的文件元素是不定长字符行，而记录文件的文件元素是记录，并且所有记录的长度是固定的。因此在建立一个新的记录文件时，必须设置其记录的长度。通常可以用两种设置方式：在 SET 语句或在 OPEN 语句中用附加可选项 recsize 设置。用 SET 语句设置记录长度的句型为：

SET #〈通道号〉:recsize〈整数表达式〉

用 OPEN 语句设置记录长度的句型前面已介绍，这里不再赘述。

当要对已有的记录文件进行插入、追加、引用等操作时，必须了解其记录长度。这时应用下面的句型询问：

ASK #〈通道号〉:recsize〈变量〉

记录长度一律以字节计。记录中的每个数值数据占8个字节，按浮点形式存储；字符串数据中的每个字符占一个字节。

3. 文件指针的移动

文件一经打开，系统便会设置一个记录计数器，即文件指针指向即将读写的记录。以"output"方式打开文件时，文件指针指向文件尾；以"input"方式打开文件时，文件指针指向文件首。然后每进行一次读写操作，记录计数器便自动加一，指向下一个记录，从而形成顺序存取方式：

True BASIC 允许修改记录文件的记录计数器，改变存取的顺序关系，形成记录文件的随机存取方式。修改记录计数器的句型是

SET #〈通道号〉:record〈记录号表达式〉

SET #〈通道号〉:POINTER { begin / same / next / end }

必要时还可以用 ASK 语句询问文件指针的内容。句型为

ASK #〈通道号〉:record〈变量〉

9.3.2 记录文件的存取

True BASIC 用 WRITE 与 READ 语句对记录文件进行存取。

1. WRITE 语句

WRITE 语句的基本句型为

WRITE #〈通道号〉:〈表达式表〉

当要对记录文件执行 WRITE 操作时，必须保证文件已经处于以"output"或"outin"方式打开的状态。同时，还要根据需要确定以顺序方式还是以随机方式进行存取操作。

例 9.3.1 建立例9.1.3所示记录文件。

该记录文件是一个简化的劳资文件。它所存储的资料由职工的姓名、年龄、工资三部分组成。在建立该记录文件之前，首先要确定记录的长度：

姓名：按汉字拼音，每个名字约占10个字符。

年龄：数值数据，8个字节。

工资：数值数据，8个字节。

由于记录是定长的，考虑上述三种数据，取其中最长者即10个字节，作为本记录文件的记录长度。

程序如下：

```
!
! example 9.3.1
!
OPEN #5:name "ex931.rec", create new,organization record, recsize 10
DO while more data
    READ name $ ,age,sal
    WRITE #5:name $ ,age, sal
LOOP
CLOSE #5
DATA "Zhang 3",50,313.50
DATA "Li    4",50,258.60
DATA "Wang  5",47,166.20
END
```

执行该程序后将会建立一个名为"ex931.rec"的记录文件，当在 DOS 命令状态下观看其内容时，将看到

```
C:\ZJW>type ex931.rec
TR
          Sli      4       N@J         N@p)
Swang  5       N@G              N@dFffff'
```

这是因为 True BASIC 以扩展的二～十进制码来存储记录文件，这种码不象 ASCII 码可以直接识别。

2. READ 语句

READ 语句的基本句型为：

```
READ #〈通道号〉:〈变量表〉
```

当要对记录文件执行 READ 操作时，必须保证文件已经处于以"input"或"outin"方式打开的状态。

例 9.3.2 读例9.3.1所建立的记录文件。

```
!
! example 9.3.2
!
OPEN #6:name "ex931.rec",organization record
DO while more #6
    READ #6:name$ ,age,sal
    PRINT name$ ,age,sal
LOOP
CLOSE #6
END
```

执行上述程序后,将在命令窗内显示:

```
Ok. run
Zhang 3        50          313.5
Li    4        52          258.6
Wang  5        47          166.2
Ok.
```

应当注意,READ 语句中的变量类型要与记录文件中对应记录的数据类型一致。

上面是一个按顺序方式读记录文件的例子。下面再举一个按随机方式读记录文件的例子。

例 9.3.3 只读“ex931.rec”中的姓名及年龄。

```
!
! example 9.3.3
!
OPEN #7:name "ex931.rec",organization record
DO while more #7
    READ #7:name$ ,age
    SET #7:pOINTER next
    PRINT name$ ,age
LOOP
CLOSE #7
END
```

程序执行结果:

```
Ok. run
Zhang 3        50
Li    4        52
Wang  5        47
Ok.
```

3. WRITE 语句与 READ 语句的扩展句型

MAT WRITE #〈通道号〉:〈数组名表〉

MAT READ #〈通道号〉:〈数组名表〉

9.3.3 多数据项记录的拼装与分解

由例9.3.1～例9.3.3可以看出，采用单数据项记录的记录文件，有如下缺点：

(1) 所有记录都占用相同的存储空间，造成存储空间的浪费。

(2) 记录号不自然。如文件“ex931.rec”中，第二个职工的姓名的记录号为“4”。这样容易出错。

解决的办法是将所有数据项都转换成字符串，然后按定长压缩连接成一个字符串，使用时，将拼装成的字符串分解。

例 9.3.4 用多数据项记录存取记录文件“ex931.rec”中的数据。

首先确定字符串的长度，姓名——7，年龄——8，工资——8，共23字节，如下图所示。

1　　　7	8　　　　15	16　　　　23
姓　名	年　龄	工　资

程序如下：

```
!
! example 9.3.5
!
DEF pack $ (hame $ ,age,sal)                !拼接函数
     LET pack $ =name $  & num $ (age) & num $ (sal)
END DEF
!
OPEN  #8:name"ex935.rec", create new, organization record, recsize 23
DO while more data
    READ name $ ,age,sal
    WRITE  #8:pack $ (name $ ,age,sal)
LOOP
CLOSE  #8
DATA "Zhang 3",50,313.50
DATA "Li     4",52,258.60
DATA "Wang   5",47,166.20
END
```

例 9.3.5 取出“ex935.rec”中的第二个记录。

```
!
! example 9.3.6
!
```

```
OPEN #9:name "ex935.rec",organization record
SET #9:record 2
READ #9:rec $
CALL unpack(rec $ ,name $ ,age,sal)
PRINT name $ ,age,sal
CLOSE #9
END
!
SUB unpack(rec $ ,name $ ,age,sal)
    LET name $ =rec $ [1:7]
    LET age=num(rec $ [8:15])
    LET sal=num(rec $ [16:23])
END SUB
```

程序的执行结果如下：

```
Ok. run
Li      4       52          258.6
Ok.
```

9.4 字节文件操作

9.4.1 字节文件的存取特点

字节文件既不象正文文件那样要以“行”为单位进行存取，也不象记录文件那样要求以“记录”为单位进行存取，而是可以以任意长度的字节(字符串)进行存取。

1. 写字节文件

向已打开的字节文件中写数据，只有一种写语句：

```
WRITE #〈通道号〉:〈字符串表〉
```

例 9.4.1

```
!
! example 9.4.1
!
OPEN #6:name "ex941.byt",organization byte,access outin,create new
FOR i=1 to 5
    LET a=int(10 * rnd)                !产生一个10以内的随机数 a
    LET b $ =repeat $ ("*",a)          ! 产生 a 个"*"
    WRITE #6:b $
```

```
NEXT i
CLOSE #6
END
```

下面是上述程序某次执行后，在文件“ex941.byt”中的存储内容：

```
******************************
```

这说明：

(1) 字节文件是以 ASCII 码存储的，是可以直接显示的。

(2) WRITE 语句每次写入的是由随机数 a 确定的不定长字节串。

(3) 最后存入文件的是一个字节流。每个字符占一个字节。

2. 读字节文件

由于允许以任意长度读字节文件中的数据，而且每次的长度可以不同，所以要从字节文件中读出字符串时，应同时指出所读字符串的长度。因此，读字节文件语句的句型为

```
READ #〈通道号〉,bytes〈字节数〉:〈字符串变量表〉
```

或采用如下一对语句

```
SET #〈通道号〉:recsize〈字节数〉
READ #〈通道号〉:〈字符串变量表〉
```

例 9.4.2 从“ex941.byt”文件中读字符，组成一种三角形图案。

```
!
! example 9.4.2
!
OPEN #8:name "ex941.byt",organization byte,access outin
ASK #8:fILESIZE size                    ! 测文件长度
ASK #8:rECORD rec                       ! 测文件指针位置
LET i=1
DO while(size-rec+1)=i)
   READ #8,bytes i:b $
   PRINT b $
   ASK #8:rECORD rec
   LET i=i+1
LOOP
CLOSE #8
END
```

执行结果：

```
Ok. run
*
* *
* * *
* * * *
* * * * *
* * * * * *
* * * * * * *
* * * * * * * *
Ok
```

由此可以说明：

(1) 每个 READ 语句中必须指出将要从流式字节串中读出的字节数。这个字节数也决定了文件指针将要后移的距离。

(2) READ 语句中的变量不可读空。因此上述程序中每次读文件之前要先询问一次文件指针的位置，以计算(WHILE 语句中)所剩下的字节数还够不够读一次。

9.4.2 字节文件应用举例

如果把记录文件比做格律诗，把正文文件比做自由体诗，则可以把字节文件看作写了字的纸。它既可以是诗，也可以是词，还可以是散文、小说、书信或者小学生练字时写下的一些字。它是一种流式文件，人们可以随心所欲地从中读取一串任意长度的字符串，也可以随心所欲地向它写入一串任意长度的字符串。任何文件都可以看作是字节文件，不管它是哪种程序创建的；用字节文件的写语句也可以创建其它组织的文件。这一灵活性，提供了比记录文件及正文文件更方便的文件编辑、复制功能。

例 9.4.3 用字节文件创建例9.1.2给出的记录文件。请与例9.3.6中的程序比较。

```
!
! example 9.4.3
!
INPUT prompt "file name:"filename $
OPEN #8:name filename $ ,create new, organization byte
DO while more data
    READ name $ ,age $ ,sal $
    LET n $ =(name $ & repeat $ (" ",10))(1:10)
    LET a $ =(age $ & repeat $ (" ",2))(1:2)
    LET s $ =(sal $ & repeat $ (" ",6))(1:6)
    WRITE #8:n $ ,a $ ,s $
LOOP
CLOSE #8
DATA "Zhang 3",50,313.50
DATA "Li    4",52,258.60
DATA "Wang  5",47,166.20
```

END

执行上述程序后，将在其所建立的字节文件中以如下格式存储了一个劳资文件。

```
C:\ZJW>type bbb. byt
Zhang   3        50313. 50Li      4        52258. 60Wang        5        47166. 20
```

这也是一个字节流。

例 9.4.4 从程序9.4.3所建立的字节文件中读出数据。

关键在于按写入时的各数据的长度依次读出各数据。

程序如下：

```
!
! example 9.4.4
!
INPUT prompt "file name:":filename $
OPEN #8:name filename $ ,create old,organization byte
DO while more #8
   READ #8:name $ ,age $ ,sal $
   PRINT name $ ,age $ ,sal $
LOOP
CLOSE #8
END
```

程序执行结果如下：

```
Ok. run
file name:bbbbb. byt
Zhang   3          50          313. 50
Li      4          52          258. 60
Wang    5          47          166. 20
Ok.
```

9.5 文件管理

这里说的文件管理指：

- 显示文件目录(查询文件名)
- 修改文件名
- 文件复制(拷贝)
- 文件删除
- 文件内容比较
- 打印(显示)文件内容

等对文件进行的"整体操作"。

文件管理可以在两种状态下进行：操作系统 DOS 的命令状态下及 True BASIC 的命令状态下进行。

表9.2给出了一些常用的文件管理命令。

表 9.2

状态	命令(语句)格式	功　能
DOS	DIR/W	横向列出全部目录项
	DIR	纵向列出全部目录项
	DIR *〈后缀名〉	列出带指定后缀名的目录项
	DIR〈文件名〉.*	列出带指定文件名的目录项
	DEL〈文件标识符〉	删除指定文件
	DEL *〈后缀名〉	删除带指定后缀的所有文件
	DEL〈文件名〉.*	删除带指定文件名的所有文件
	DEL *.*	删除所有文件
	RENAME〈旧文件标识符〉,〈新文件标识符〉	文件换名
	COPY A:〈原文件标识符〉,B:〈目标文件标识符〉	将 A 盘中一文件复制为 B 盘一文件并换名
	COPY A:〈文件标识符〉,B:	将 A 盘中指定文件复制到 B 盘
	COPY A:*.* B:	将 A 盘中所有文件复制到 B 盘
	DISKCOPY A: B:	将 B 盘格式化并将 A 盘全部内容复制
	COMP〈文件标识符1〉〈文件标识符2〉	比较两文件是否相同
	DISKCOMP A: B[/1]	比较两磁盘内容是否相同
	TYPE〈文件标识符〉	显示文件内容
True BASIC	FILES	显示文件目录
	UNSAVE〈文件标识符〉	删除指定文件
	ERASE #〈通道号〉	将与指定通道连接的文件置空

习　题

9.1.1　文件与数组有何异同？

9.1.2　试总结 DOS 的文件的命名规则。

9.1.3　文件的顺序存取与随机存取方式下，文件指针各是如何移动的？哪一种是被系统移动的，哪一种是被用户程序移动的？

9.1.4　记录文件可不可以被顺序存取？

9.1.5　比较正文文件、记录文件、字节文件的异同，并举例说明。

9.1.6　在文件操作过程中，什么时候打开文件？什么时候关闭文件？

9.1.7　打开文件的作用是什么？

9.1.8　文件存、取时的大致步骤各是什么？要注意哪几点？

9.2.1 指出写正文文件 ABCD.EFG 程序中的错误并改正。

```
OPEN #0, name ABCD.EFG, create new organization text
READ A$,B,C,
PRINT A$,B,C
END
DATA 北京,01-2595131, 100084
```

9.2.2 指出下面读正文文件 ABCD.EFG 中的错误。

```
OPEN #3333:name "ABCD.EFG", organization text
DO
INPUT A,B,C
PRINT A,B,C
LOOP
END
```

9.2.3 用正文文件写一个 True BASIC 程序。

9.3.1 在记录文件中,用什么确定对哪个记录进行存取?

9.3.2 建立一个存储友人电话号码的记录文件。并且设计一个可以按人名查询其电话号码的程序。

9.3.3 请设计一个对记录文件进行插入、删除的算法。

9.4.1 将磁盘上的任意一个 True BASIC 源程序文件作为字节文件,统计其中 LET 语句的个数。

附录A 编辑键一览表

键符号	名 称	功 能
F1		将光标移到编辑窗口。
F2		将光标移到背景窗口。
F3	FIND 键	从光标所在处开始搜索某字或字符。
F4	MARK 键	设置/解除行标记。
F5	COPY 键	将有标记的行模块复制到另一处。
F6	MOVE 键	将有标记的行模块移动到另一处。
>		将有标记的行模块右移一格。
<		将有标记的行模块左移一格。
F7		再现被 New Delete 删除的行模块,或上次的命令。
F9	RUN 键	执行当前程序。
Del	DELETE 键	·删除光标所指字符; ·光标在行标志处时,删除当前行; ·删除有标记的行模块。
←	Backspace 键	·删除光标左边一个字符; ·光标在行标志处时,将当前行连接到上一行尾。
↲	Return 键	·将光标以右部分移至下一行行首; ·光标在行标志处时,插入一空行。
Ins	Insert 键	插入或重写。
Ctrl-F,Ctrl-B,Ctrl-E		改变前景、后景和边缘的颜色。

注:·编辑键的操作多以光标位置作参考。

·以上是指在长城 0520 及 IBM PC 上的操作,在其它机器上应参阅有关手册。

附录B True BASIC系统命令一览表

B.1 文件操作

命令名称	用途
FILES	显示 True BASIC 文件目录。
NEW	清除当前内存文件(程序)。
OLD	装入一磁盘文件。
SAVE	当前内存文件存盘。
REPLACE	在磁盘上重新写入一同名文件。
UNSAVE	删除一磁盘文件。

B.2 编辑

命令名称	用途
CHANGE	改换一字或一短语。
COPY	在文件中复制一部分到别处。
DELETE	删除文件中的一部分。
DO FORMAT	格式当前文件。
EDIT	对文件的一部分进行限制性编辑。
INCLUDE	在当前文件中光标所在行后面插入一磁盘文件。
KEEP	删除当前文件中指出部分以外的程序行。
LOCATE	显示含有某字或短语的程序行。
MARK	标记指定的程序块。
MOVE	移动文件的一部分到别处。
TRY	搜索某一字或短语,等待用户回答后决定是否更改。 ·N:本次不改换。 ·Y:本次改换。 ·Q:不改,并退出修改。

B.3 查错

命令名称	用途
BREAK	插入断点。
CONTINUE	断点后继续。

B.4 更改行号

命令名称	用途
DO NUM	加行号。
DO RENUM	分配新行号。
DO UNNUM	去掉行号。

B.5 其它

命 令 名 称	用　　途
BYE	退出 True BASIC。
COMPILE	编辑当前文件。
DO	在一文件上运行预处理程序。
HELP	提供帮助。
KEY	重新定义键的功能。
LIST	打印当前文件清单。
MOD	模式设置。
RUN	执行当前文件。
TO	移光标到某行。

注：①所有 True BASIC 命令可以只写前三个字母，系统便可以识别。但是不能在超过三个字符时写错。如 REPLACE 可以写成 REP 或 REPL，REPLA，REPLAC 等，但不能写成 REPI，REPLQ，REPLAX 等。

②你可以在一行上键入一些用分号隔开的命令。True BASIC 将作为批命令执行它们，直到全部执行完为止。例如，Ok. old flash;run;old sort; run。执行完这些命令或执行到有错命令时，True BASIC 将结束执行，返回命令状态显示出"Ok"提示。

附录C True BASIC 语法一览表

C.1 简单语句

语 句	说 明
ASK MARGIN〈变量〉	询问当前显示屏幕边界。
ASK ZONEWIDTH〈变量〉	询问当前显示区域宽度。
DATA〈数据 1〉,〈数据 2〉,……	建立程序数据区。
END	终止程序运行。
GET KEY〈数字变量〉	读取键代码。
INPUT〈变量 1〉,〈变量 2〉,……	由键盘向变量传送数据。
INPUT PROMPT〈串表达式〉:〈变量 1〉,〈变量 2〉,……	带提示的键盘输入。
LINE INPUT〈串变量 1〉,〈串变量 2〉,……	由键盘输入带空格或逗号的一行字符串。
LINE INPUT PROMPT〈串表达式〉:〈串变量 1〉,〈串变量 2〉,……	带提示的键盘输入一行字符串。
LET〈变量 1〉,〈变量 2〉,……=〈表达式〉	将一个表达式的值赋给几个变量。
LET a$[i:j]=〈串表达式〉	用串表达式的值改变串 a$ 中第 i 到 j 个字符
OPEN #〈通道号〉,PRINTER PAUSE〈秒数〉	打开打印机通道。通道号为正文文件通道号。 使机器暂停指定秒数的一段时间。
PLAY〈音乐字符串表达式〉	演奏乐曲。
PRINT〈表达式 1〉,〈表达式 2〉,……	按分区格式显示。
PRINT〈表达式 1〉;〈表达式 2〉,……	按紧凑格式显示。
PRINT #〈通道号〉:〈表达式 1〉,〈表达式 2〉,……	按分区格式打印。
PRINT #〈通道号〉:〈表达式 1〉;〈表达式 2〉,……	按紧凑格式打印。
PRINT USING〈格式串〉:〈表达式 1〉,〈表达式 2〉,……	按自选格式打印。
PRINT #〈通道号〉,USING〈格式串〉:〈表达式 1〉,〈表达式 2〉,……	按自选格式打印。
RANDOMIZE	捕捉随机数序列。
READ〈变量 1〉,〈变量 2〉,……	由本程序单位数据区向变量传送数据。
RESTORE	恢复本程序单位的数据指针。

续表

语　　句	说　　明
SET MARGIN〈数字表达式〉	设置屏幕边界。
SET MOD〈串表达式〉	设置模式。
SET ZONE WIDTH〈数字表达式〉	设置显示区宽度。
STOP	暂停程序运行。
SOUND〈频率〉,〈秒数〉	使扬声器发声。

注：上述语法描述中的省略号(……)表示该语句中的项可以多于或少于所列出的项数；光标号(〈 〉)中的内容由用户给出。

C.2　循环结构

控制结构或语句	说　　明
DO …… …… }〈语句块〉 …… LOOP	无条件循环。
DO WHILE〈条件〉 …… …… }〈语句块〉 …… LOOP	当型循环。
DO …… …… }〈语句块〉 …… LOOP UNTIL〈条件〉	直到型循环。
DO WHILE〈条件〉 …… …… }〈语句块〉 …… LOOP UNTIL〈条件〉	当——直到型循环。
EXIT DO	跳出 DO 循环。
FOR〈循环变量〉=〈初值〉TO〈终值〉STEP〈步长〉 …… …… }〈语句块〉 …… NEXT〈循环变量〉	FOR 循环。
EXIT FOR	跳出 FOR 循环。

C.3 选择结构

控制结构或语句	说　　明
IF〈条件〉THEN〈语句〉	
IF〈条件〉THEN …… ……〈语句块〉 …… END IF	
IF〈条件〉THEN〈语句〉ELSE〈语句〉	

IF〈条件〉THEN
……
……〈行模块 1〉
……
ELSE
……
……〈行模块 2〉
……
END IF	

IF〈条件 1〉THEN
……
……〈行模块 1〉
……
ELSE IF〈条件 2〉THEN
……
……〈行模块 2〉
……
⋮
ELSE
……
……〈行模块 n〉
……
END IF
	① 可以包含多个 ELSE IF 块；
	② 可以包含零个或一个 ELSE 块。

SELECT CASE〈表达式〉 CASE〈测试 1.1〉,〈测试 1.2〉,…… …… ……〈行模块 1〉 …… CASE〈测试 2.1〉,〈测试 2.2〉,……	测试是针对测试表达式的。 测试可以有下列一种形式： • 常数； • 〈低值〉TO〈高值〉； • is〈关系操作〉

续表

控制结构或语句	说　　明
…… ……〉〈行模块 2〉 …… ⋮ CASE ELSE …… ……〉〈行模块 n〉 …… END SELECT	

C.4　矩阵语句

语　　句	说　　明
MAT〈数组 1〉,〈数组 2〉,……=〈数组〉	数组赋值。
MAT〈数组 1〉,〈数组 2〉,……=〈表达式〉	用一个表达式给 n 个数据赋值。
MAT〈数组 1〉=〈数组 2〉+〈数组 3〉	数组加。
MAT〈数组 1〉=〈数组 2〉*〈数组 3〉	数组乘。
MAT〈数组 1〉=〈常数〉*〈数组 2〉	数字乘数组。
MAT INPUT〈数组 1〉,〈数组 2〉,……	数组键盘输入。
MAT INPUT〈数组〉(?)	数组输入并调整。
MAT INPUT PROMPT〈串表达式〉:〈数组 1〉,〈数组 2〉,……	带提示的数组键盘输入。
MAT LINE INPUT〈数组 1〉,〈数组 2〉,……	从键盘读入足够的行给一数组的各元素。
MAT LINE INPUT PROMPT〈串表达式〉:〈数组 1〉,〈数组 2〉,……	带提示的数组整行键盘输入。
MAT PRINT〈数组 1〉,〈数组 2〉,……	以分区格式显示各数组元素。
MAT PRINT〈数组 1〉,〈数组 2〉,……	以紧凑格式显示各数组元素。
MAT PRINT USING〈格式串〉:〈数组 1〉,〈数组 2〉,……	以自选格式显示各数组元素。
MAT READ〈数组 1〉,〈数组 2〉,……	数组读。

C.5 程序单位

控制语句或结构	说　　明
CALL〈子程序名〉(〈实参 1〉,〈实参 2〉,……)	调用子程序。
CHAIN〈文件名〉	连接指定文件。
CHAIN〈文件名〉WITH(〈串表达式〉)	带参数传递的文件连接。
CHAIN〈文件名〉,RETURN	连接文件执行完后返回原来程序。
CHAIN〈文件名〉WITH(〈串自变量〉),RETURN	
DECLARE DEF〈函数名 1〉,〈函数名 2〉,……	函数命名。
DEF〈函数名〉(〈参数 1〉,〈参数 2〉,……)=〈表达式〉	单行函数定义。
DEF〈函数名〉(〈参数 1〉,〈参数 2〉,……) …… …… }〈语句块〉 …… LET〈函数名〉=〈表达式〉 …… …… }〈语句块〉 …… END IF	多行函数定义。
EXIT DEF	中间返回。
EXIT SUB	中间返回。
LIBRARY"〈文件名 1〉","〈文件名 2〉",……	指定库文件。
PROGRAM〈文件名〉	被连接文件开始。
PROGRAM〈文件名〉(〈串变量〉)	带参数的被连接文件开始。
SUB〈子程序名〉(〈参数 1〉,〈参数 2〉,……) …… …… }〈语句块〉 …… END SUB	定义子程序。

C.6 图形处理

语　　句	说　　明
ASK BACK〈变量〉	询问背景色号。
ASK BACK〈串变量〉	询问背景色名。
ASK BACKGROUND COLOR〈变量〉	同 ASK BACK〈变量〉。
ASK BACKGROUND COLOR〈串变量〉	同 ASK BACK〈串变量〉。
ASK COLOR〈变量〉	询问前景色号。

续表

语　　句	说　　明
ASK COLOR〈串变量〉	询问前景色名。
ASK CURSOR〈串变量〉	询问光标状态为"ON"还是"OFF"。
ASK CURSOR x,y	询问光标位置。
ASK MAX CURSOR x,y	询问当前窗口大小。
ASK MAX COLOR〈变量〉	询问计算机可以使用几种色。
ASK SCREEN〈左〉,〈右〉,〈底〉,〈顶〉	询问当前窗口屏幕坐标。
ASK WINDON〈左〉,〈右〉,〈底〉,〈顶〉	询问当前窗口坐标。
BOX AREA〈左〉,〈右〉,〈底〉,〈顶〉	画一个盒子并用当前色涂满。
BOX CIRCLE〈左〉,〈右〉,〈底〉,〈顶〉	同 BOX ELLIPSE。
BOX CLEARE〈左〉,〈右〉,〈底〉,〈顶〉	抹去一个盒子。
BOX ELLIPSE〈左〉,〈右〉,〈底〉,〈顶〉	画一个盒子的内切椭圆。
BOX KEEP〈左〉、〈右〉,〈底〉,〈顶〉IN〈串变量〉	保存一部分显示到一个串变量中。
BOX LINES〈左〉,〈右〉,〈底〉,〈顶〉	画一个盒子边线。
BOX SHOW〈串变量〉AT〈左〉,〈底〉	再次显示存于串变量中的图像。
BOX SHOW〈串变量〉AT〈左〉,〈底〉USING〈串变量〉	存储图像与屏幕图像组合,〈串变量〉为"and","or"或"xor"。
BOX SHOW〈串变量〉AT〈左〉,〈底〉USING〈数字〉	
CLEAR	使整个窗口为当前背景色。
CLOSE #〈窗口号〉	关闭一个窗口。
DRAW〈图画〉	调用一个图画。
DRAW〈图画〉WITH〈变换〉	调用一个图画,并进行变换。
FLOOD x,y	转换一个点的前景色,并扩展到同色的邻近像素点上。
GET MOUSE : x,y	用鼠标串在点(x,y)设置光标。
GET POINT : x,y	在坐标(x,y)设置光标。
MAT PLOT AREA :〈数字数组〉	用坐标数组数据画实体。
MAT PLOT LINES :〈数字数组〉	用坐标数组数据画线。
MAT PLOT POINTS :〈数字数组〉	用坐标数组数据画点。
OPEN #〈窗口号〉: SCREEN〈左〉,〈右〉,〈底〉,〈顶〉	打开一个窗口。

语　　句	说　　明
PICTURE〈图画名〉x(〈参数 1〉,〈参数 2〉,……) …… ……〉〈语句块〉 …… END PICTURE	定义一个图画。 语句块可以不包括。 SET WINDOW 或 OPEN SCREE 语句。
PLOT x,y	
PLOT x,y	
PLOT x1,y1;x2,y2;……	画点。
PLOT AREA : x1,y1,x2,y2;……	用当前色填充区域。
PLOT LINES : x1,y1,x2,y2;……	画线。
PLOT POINTS : x1,y1,x2,y2;……	画点。
PLOT TEXT, AT x,y :〈串表达式〉	放标题到图形上。
SET BACK〈表达式〉	按色号设置背景色。
SET BACK〈串表达式〉	按色名设置背景色。
SET BACK GROUND COLOR〈表达式〉	按色号设置背景色。
SET BACK GROUND COLOR〈串表达式〉	按色名设置背景色。
SET COLOR〈表达式〉	按色号设置前景色。
SET COLOR〈串表达式〉	按色名设置前景色。
SET CURSOR〈行〉,〈列〉	在屏幕任意位置上定义坐标。
SET CURSOR〈串表达式〉	设置/删除坐标。串表达式可以是“off”或“on”。
SET CURSOR x,y	给当前窗口设置坐标。
SET WINDOW〈左〉,〈右〉,〈底〉,〈顶〉	设置窗口。
WINDOW #〈窗口号〉	窗口间转换。

C.7 文件处理

语　　句	说　　明
ASK #〈通道号〉: ACCESS〈串变量〉	询问文件操作方式(“INPUT”,“OUT PUT”或“OUTIN”)。
ASK #〈通道号〉: FILESIZE〈变量〉	询问文件尺寸(正文、字节文件为字节数,记录文件为记录数)。
ASK #〈通道号〉: MARGIN〈变量〉	询问正文文件范围(记录、字节文件为“0”)。
ASK #〈通道号〉: NAME〈串变量〉	询问文件名。
ASK #〈通道号〉: ORGANIZATION〈串变量〉	询问文件组成形式(“TEXT”,“RECORD”或“BYTE”)。

续表

语　　句	说　　明
ASK #〈通道号〉：POINTER〈串变量〉	询问文件指针位置（“BEGIN”，“MIDDLE”或“END”）。
ASK #〈通道号〉：RECORD〈变量〉	询问文件当前指针位置（字节数或记录数）。
ASK #〈通道号〉：RECSIZE〈变量〉	询问记录文件大小（正文文件和字节文件为零）。
ASK #〈通道号〉：ZONEWIDTH ＜	询问正文文件的范围（记录文件和字节文件为零）。
CLOSE #〈通道号〉	关闭一个通道。
ERASE #〈通道号〉	删除 ·文件内容。
INPUT #〈通道号〉：〈变量1〉，〈变量2〉，……	从打开的文件中读取数据。
LINE INPUT #〈通道号〉：〈变变量1〉，〈串变量2〉，……	从打开文件中读取整行数据。
MAT LINE INPUT #〈通道号〉：〈数组1〉，〈数组2〉，……	从打开文件中读取整行数据到数组。
OPEN #〈通道号〉：NAME〈文件名〉， ACCESS INPUT OUTPUT OUTIN	 打开文件通道，只读。 只写。 读/写。
CREATE NEW OLD NEWOLD	建立新文件。 使用旧文件。 使用旧文件/建新文件。
ORGANIZATION TXEX RECORD BYTE	正文文件。 记录文件。 字节文件。
READ #〈通道号〉：〈变量1〉，〈变量2〉，……	从记录文件读取数据。
MAT READ #〈通道号〉：〈数组1〉，〈数组2〉，……	从文件中读数组。
MAT WRITE #〈通道号〉：〈数组1〉，〈数组2〉，……	向文件中写数组。
READ #〈通道号〉，BYTES	从字节文件中读数据。
RESET #〈通道号〉：BEGIN	同 SET POINTER。
RESET #〈通道号〉：NEXT	同 SET POINTER。
RESET #〈通道号〉：RECORD	同 SET POINTER。
RESET #〈通道号〉：SAME	同 SET POINTER。
SET #〈通道号〉：MARGIN〈数字表达式〉	设置正文文件范围。
SET #〈通道号〉：POINTER BEGIN	移动指针到文件头。
SET #〈通道号〉：POINTER END	移动指针到文件尾。

续表

语　　句	说　明
SET #〈通道号〉：POINTER NEXT	移动指针到下一记录。
SET #〈通道号〉：POINTER SAME	置指针到上一条记录。
SET #〈通道号〉：RECORD〈数字表达式〉	移动记录指针。
SET #〈通道号〉：RECSIZE〈数字表达式〉	设置记录文件长度。
SET #〈通道号〉：ZONEWIDTH〈数字表达式〉	设置正文文件范围。
UNSAVE〈文件名〉	删除盘上指定文件
WRITE #〈通道号〉,〈表达式 1〉,〈表达式 2〉,……	将一个或多个数据写入记录文件。

注：① 本表内语句适用于任何文件类型(文本,记录,字节)。
② 通道号值为 0～999 的数字表达式。

C.8 出错处理

语句或控制结构	说　　明
CAUSE ERROR〈整数表达式〉	捕捉出错代码。
CAUSE ERROR〈整数表达式〉,〈串表达式〉	自己定义出错代码和出错原因。
CAUSE EXCEPTION〈整数表达式〉	同 CAUSE ERROR。
CAUSE EXCEPTION〈整数表达式〉,〈串表达式〉	同 CAUSE ERROR
EXIT HANDLER	从嵌套的错误处理程序中拒绝某一错误而进入子范围内的错误处理。
WHEN ERRON IN …… …… 〈保护模块〉 …… USE …… …… 〈错误处理模块〉 …… END WHEN	出错处理模块。
WHEN EXCEPTION IN …… …… 〈保护模块〉 ……	出错处理模块。

续表

语句或控制结构	说　　明
USE 　　…… 　　……〉(错误处理模块) 　　…… END WHEN	

附录D IBM-PC字符与ASCII代码对照表

True BASIC 允许在一个字符串或注释中使用机器能识别的任意字符。在本附录里，将说明 IBM-PC 的字符集以及它们的使用方法。

D.1 显示符号集

下面列出了 IBM-PC 机上的所有字符。其中第一个至第 128 个字符是标准的 ASCII 字符。这些字符在任何种类的机器上几乎是相同的。IBM-PC 机也是如此。但是，也有象 0 至 31 这些特殊符号，它们可以在屏幕上显示或在打印机上打印出来。

表中第 129 至 256 个字符是 IBM-PC 机上特有的字符。若在程序中编写了这些字符的显示和打印，那么一旦换了其它类型的机器，可能产生不同的字符。此外，这些字符在“图形”和“高分辨率”模式时可能不显示。

表 D.1

十进制码值	字符	十六进制码值	十进制码值	字符	十六进制码值
000	(null)	00	023	↨	17
001	☺	01	024	↑	18
002	☻	02	025	↓	19
003	♥	03	026	→	1A
004	♦	04	027	←	1B
005	♣	05	028	∟	1C
006	♠	06	029	◆	1D
007	(beep)	07	030	▲	1E
008	◘	08	031	▼	1F
009	(tab)	09	032	(space)	20
010	(line feed)	0A	033	!	21
011	(home)	0B	034	"	22
012	(form feed)	0C	035	#	23
013	(carriage return)	0D	036	$	24
014	♫	0E	037	%	25
015	¤	0F	038	&	26
016	►	10	039	'	27
017	◄	11	040	(	28
018	↕	12	041	)	29
019	‼	13	042	*	2A
020	¶	14	043	+	2B
021	§	15	044	,	2C
022	▬	16	045	-	2D

续表

十进制码值	字符	十六进制码值	十进制码值	字符	十六进制码值
046	。	2E	087	W	57
047	/	2F	088	X	58
048	0	30	089	Y	59
049	1	31	090	Z	5A
050	2	32	091	[	5B
051	3	33	092	\	5C
052	4	34	093	]	5D
053	5	35	094	Λ	5E
054	6	36	095	—	5F
055	7	37	096	`	60
056	8	38	097	a	61
057	9	39	098	b	62
058	:	3A	099	c	63
059	;	3B	100	d	64
060	<	3C	101	e	65
061	=	3D	102	f	66
062	>	3E	103	g	67
063	?	3F	104	h	68
064	@	40	105	i	69
065	A	41	106	j	6A
066	B	42	107	k	6B
067	C	43	108	l	6C
068	D	44	109	m	6D
069	E	45	110	n	6E
070	F	46	111	o	6F
071	G	47	112	p	70
072	H	48	113	q	71
073	I	49	114	r	72
074	J	4A	115	s	73
075	K	4B	116	t	74
076	L	4C	117	u	75
077	M	4D	118	v	76
078	N	4E	119	w	77
079	O	4F	120	x	78
080	P	50	121	y	79
081	Q	51	122	z	7A
082	R	52	123	{	7B
083	S	53	124	¦	7C
084	T	54	125	}	7D
085	U	55	126	~	7E
086	V	56	127	⌂	7F

续表

十进制码值	字符	十六进制码值	十进制码值	字符	十六进制码值
128	Ç	80	169	⌐	A9
129	ü	81	170	¬	AA
130	é	82	171	½	AB
131	â	83	172	¼	AC
132	ä	84	173	¡	AD
133	à	85	174	«	AE
134	å	86	175	»	AF
135	ç	87			(*cont.*)
136	ê	88	176	░	B0
137	ë	89	177	▒	B1
138	è	8A	178	▓	B2
139	ï	8B	179	│	B3
140	î	8C	180	┤	B4
141	ì	8D	181	╡	B5
142	Ä	8E	182	╢	B6
143	Å	8F	183	╖	B7
144	É	90	184	╕	B8
145	æ	91	185	╣	B9
146	Æ	92	186	║	BA
147	ô	93	187	╗	BB
148	ö	94	188	╝	BC
149	ò	95	189	╜	BD
150	û	96	190	╛	BE
151	ù	97	191	┐	BF
152	ÿ	98	192	└	C0
153	ö	99	193	┴	C1
154	Ü	9A	194	┬	C2
155	¢	9B	195	├	C3
156	£	9C	196	─	C4
157	¥	9D	197	┼	C5
158	Pt	9E	198	╞	C6
159	ƒ	9F	199	╟	C7
160	á	A0	200	╚	C8
161	í	A1	201	╔	C9
162	ó	A3	202	╩	CA
163	ú	A4	203	╦	CB
164	ñ	A5	204	╠	CC
165	Ñ	A5	205	═	CD
166	ª	A6	206	╬	CE
167	º	A7	207	╧	CF
168	¿	A8	208	╨	D0

续表

十进制码值	字符	十六进制码值	十进制码值	字符	十六进制码值
209	╤	D1	233	θ	E9
210	╥	D2	234	Ω	EA
211	╙	D3	235	δ	EB
212	╘	D4	236	∞	EC
213	╒	D5	237	φ	ED
214	╓	D6	238	∈	EE
215	╫	D7	239	∩	EF
216	╪	D8	240	≡	F0
217	┘	D9	241	±	F1
218	┌	DA	242	≥	F2
219	█	DB	243	≤	F3
220	▄	DC	244	⌠	F4
221	▌	DD	245	⌡	F5
222	▐	DE	246	÷	F6
223	▀	DF	247	≈	F7
224	α	E0	248	°	F8
225	β	E1	249	•	F9
226	Γ	E2	250	·	FA
227	π	E3	251	√	FB
228	Σ	E4	252	Π	FC
229	σ	E5	253	Z	FD
230	μ	E6	254	■	FE
231	τ	E7	255	(blank)	FF
232	Φ	E8			

D.2 获得键的符号集

表 D.2 是 IBM-PC 机键盘的样图及各种可能的击键所产生的代码。这些码可由 GET KEY 语句得到。

注意，若想把码 0 至 255 中的任意数值所对应的字符显示出来，可以用 Alt 键和右手边的键盘上的数字键来实现。操作方法是在按住 Alt 键的同时按数字键。然后放开 Alt 键。这时相应的字符就出现在屏幕上。这里只允许使用数字键，而键盘上的最上一行的数字键是不起作用的。

表中，每个键下面的四行数字依次表示：常用码、上档键、控制码和交替码。表中在某些键下面使用了下面几种符号：

— 表示没有字符；

· 表示暂停输出；

↑ 表示靠不住(255,256)；

≠ 表示屏幕打印；

§ 表示使用 Alt 键与数字键，产生 0—255 之间的任何字符。

表 D.2

F1	F2	ESC	! 1	@ 2	# 3	$ 4	% 5	^ 6	& 7	* 8	(9	) 0	_ -	+ =	←	Num Lock	Scroll Lock
315	316	27	49	50	51	52	53	54	55	56	57	48	45	61	8	—	—
340	341	27	33	64	35	36	37	94	38	42	40	41	95	43	8	—	—
350	351	27	—	259	—	—	—	30	—	—	—	—	31	—	127	—	STOP
360	361	—	376	377	378	379	380	381	382	383	384	385	386	387	—	—	—

F3	F4	⇆	Q	W	E	R	T	Y	U	I	O	P	{ [	}]	7 Home	8 ↑	9 PgUp	-
317	318	9	113	119	101	114	116	121	117	105	111	112	91	93	327	328	329	45
342	343	271	81	87	69	82	84	89	85	73	79	80	123	125	55	56	57	45
352	353	—	17	23	5	18	20	25	21	9	15	—	27	29	375	—	388	—
362	363	—	255	273	274	256 ↑	276	277	278	279	280	281	—	—	§	§	§	—

F5	F6	Ctrl	A	S	D	F	G	H	J	K	L	: ;	" '	~ \	↵	4 ←	5	6 →
319	320	—	97	115	100	102	103	104	106	107	100	59	39	96	13	331	—	333
344	345	—	65	83	68	70	71	72	74	75	76	58	34	126	13	52	53	54
354	355	—	1		4	6	7	8	10	11	12	—	—	—	10	371	—	372
364	365	—	286	287	288	289	290	291	292	293	294	—	—	—	—	§	§	§

F7	F8	⇑	| \	Z	X	C	V	B	N	M	<	>	? /	⇑	PnSc	1 End	2 ↓	3 PgDn
321	322	—	92	122	120	99	118	98	110	109	44	46	47	—	42	335	336	337
346	347	—	124	90	88	67	86	66	78	77	60	62	63	—	≠	49	50	51
356	357	—	28	26	24	3	22	2	14	13	—	—	—	—	—	373	—	374
366	367	—	—	300	301	302	303	304	305	306	—	—	—	—	—	§	§	§

F9	F10	Alt	(空格)	Caps Lock	0 Ins	Del	+
323	324	—	32	—	338	339	43
348	349	—	32	—	48	46	43
358	359	—	32	—	—	—	—
368	369	—	32	—	§	—	—

D.3 打印机符号集

有许多打印机所打印的字符与屏幕显示的不一致。例如：屏幕上显示为横线的字符，在打印机上被打印成方块。IBM-PC 机为打印机提供了三种字符集。两种“图形”方式和一种“点阵”方式。关于打印机上的字符集请看 IBM-PC 的技术参考手册。

Document generated by Anna’s Archive around 2023-2024 as part of the DuXiu collection (https://annas-blog.org/duxiu-exclusive.html).

Images have been losslessly embedded. Information about the original file can be found in PDF attachments. Some stats (more in the PDF attachments):
{
  "filename": "MTE0MDk1NDguemlw",
  "filename_decoded": "11409548.zip",
  "filesize": 37489059,
  "md5": "0549f9529aac60d1f97c66ab9a8ece84",
  "header_md5": "28f46c6af9e9a8e141571164e46009ac",
  "sha1": "995f831766183b127c4ebe7a534e4226f0e9cacd",
  "sha256": "ace2f16f7a67b89b4b71b7efe2da2a0a364e3af699907f4119aef71552709998",
  "crc32": 2132025290,
  "zip_password": "julian",
  "uncompressed_size": 39313584,
  "pdg_dir_name": "TRUE BASIC\u2502\u2560\u2568\u2265\u2554\u03a6\u255d\u255e
\u00fa\u00bf\u2561\u250c\u2562\u25a0\u2591\u00b5\u00fa\u2310_11409548",
  "pdg_main_pages_found": 326,
  "pdg_main_pages_max": 326,
  "total_pages": 341,
  "total_pixels": 2224313280,
  "pdf_generation_missing_pages": false
}